Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo de fin de Grado:

Diseño y desarrollo de un cliente y un servidor en JavaScript para gestionar batallas y campeonatos entre agentes inteligentes (JSWARS).

Autor: Marcos Pérez Ferro
Tutor: Carlos Linares López

Febrero 2016

Agradecimientos

A ti, ***Laura***

por tu infinita paciencia, sacrificio y

tu apoyo incondicional.

Sin ti, nunca habría llegado hasta aquí…

A ti, ***Luis***

por ser el mejor amigo y compañero

que se puede tener.

Esta es solo la primera

de muchas locuras compartidas…

Abstract

JWARS is a platform for Artificial Intelligence competition, designed for closing distances between newbies and experts in the field. Its basic approach consists of the design and development of a framework that allows the user to execute simple battles with basic rules: two teams, each team's army battle to death, team surviving most time wins. Each army follows the logic of one artificial intelligence agent, the one the user creates and uploads.

Battles themselves are in real time and the whole project is hosted in a web platform with Github login, allowing users to create and upload several agents quick and easy and encouraging competition. To this purpose, the platform also allows real time tournaments between several users and its agents.

Platform is divided in two different parts: web application (server and client) which manages everything the user interacts with, and game engine, which represents and executes the battles using the agents codes the user selects.

In this project we are studying everything involved in the first part, web application, from general architecture to particular details in every subsystem taking part.

ÍNDICE

Tabla de ilustraciones 1
1 Introducción 5
2 Motivación y objetivos 7
2.1 Motivación 7
2.2 Objetivos y alcance del proyecto 8
3 Estado de la cuestión 9
3.1 Competiciones de IA 10
3.1.1 Elevator Saga 10
3.1.2 Ants AI Challenge 11
3.2 Otras competiciones 12
3.2.1 CodeFights 12
3.2.2 HackerEarth 13
4 Desarrollo 15
4.1 Análisis 15
4.1.1 Objetos del sistema 15
4.1.2 Requisitos del sistema 17
4.1.3 Casos de Uso 28
4.2 Diseño 38
4.2.1 Arquitecturas web disponibles 38
4.2.2 Arquitectura web elegida 40
5 Subsistema cliente 43
5.1 Elección de tecnologías 43
5.1.1 HTML 43
5.1.2 CSS3 43
5.1.3 JavaScript 44
5.1.4 AngularJS 44
5.1.5 KineticJS 46
5.1.6 RequireJS 46
5.1.7 Grunt 46
5.1.8 Bower 47
5.2 Arquitectura 48

5.2.1 Componentes desarrollados 49
5.2.2 Interfaces implementadas 55
5.3 Consideraciones 68
5.3.1 Posicionamiento en motores de búsqueda 68
5.3.2 Reproducción de partidas y refresco de pantalla 69
5.3.3 Recepción de partidas de forma particionada 70
6 Subsistema servidor 73
6.1 Evaluación de Tecnologías 73
6.1.1 NodeJS 73
6.1.2 MongoDB 73
6.1.3 Jshint 74
6.1.4 Express 74
6.2 Arquitectura 75
6.2.1 API Rest 75
6.2.2 Base de datos 79
6.2.3 Cola de ejecución de batallas (QueueRunner) 82
6.2.4 Cola de ejecución de torneos (TournamentRunner) 83
6.3 Consideraciones 83
6.3.1 Gestión de eventos en NodeJS 83
6.3.2 Almacenamiento en disco 89
7 Herramientas de desarrollo 91
7.1 Sistema Operativo (Debian) 91
7.2 Entorno de desarrollo integrado (IDE): WebStorm 91
7.3 Control de versiones: GIT 92
7.4 Exploradores 92
8 Planificación 93
9 Aspectos económicos 95
9.1 Coste de desarrollo 95
9.1.1 Personal requerido 95
9.1.2 Coste del personal 96
9.1.3 Costes de Hardware y Software 97
9.1.4 Costes indirectos 98
9.1.5 Resumen de costes 98

9.1.6 Presupuesto ... 99
9.2 Coste de puesta en marcha y mantenimiento ... 100
9.2.1 Personal requerido ... 100
9.2.2 Coste del personal ... 100
9.2.3 Costes de Hardware y Software ... 101
9.2.4 Resumen de costes ... 102
9.2.5 Presupuesto ... 102
10 Resultados ... 103
10.1 Pruebas objetivas ... 103
10.2 Pruebas subjetivas ... 104
11 Líneas futuras ... 107
11.1 Escalabilidad ... 107
11.2 Aumentar número de mapas disponibles ... 107
11.3 Más de 2 agentes por partida ... 108
11.4 Rankings ... 108
11.5 Potenciar funcionalidades sociales ... 108
12 Conclusiones ... 111
Bibliografía ... 113
Anexos ... 115
Anexo 1: Gantt Planificación ... 117
Anexo 2: Presupuesto desarrollo ... 119
Anexo 3: Presupuesto puesta en marcha ... 121

TABLA DE ILUSTRACIONES

Ilustración 1 Aspecto de Elevator Saga 10
Ilustración 2 Aspecto de Ants AI Challenge 11
Ilustración 3 Aspecto de CodeFight 12
Ilustración 4 Aspecto de HackerEarth 13
Ilustración 5 Ejemplo de mapa y su posible aspecto 16
Ilustración 6 Formato torneos 17
Ilustración 7 Diagrama de casos de uso en referencia al objeto Usuario 29
Ilustración 8 Diagrama de casos de uso en referencia al inicio de sesión del objeto usuario 29
Ilustración 9 Diagrama de casos de uso en referencia al objeto Agente 31
Ilustración 10 Diagrama de casos de uso en referencia al objeto Batalla 34
Ilustración 11 Diagrama de casos de uso en referencia al objeto Torneo 36
Ilustración 12 Diagrama de casos de uso en referencia al concepto Documentación 37
Ilustración 13 Diagrama de flujo aplicaciones 'Multiple page application' 38
Ilustración 14 Diagrama de flujo aplicaciones 'Single page application' 39
Ilustración 15 Flujo 'One Way Data Bindind' 45
Ilustración 16 Flujo 'Two Way Data Bindind' 45
Ilustración 17 Arquitectura del cliente 48
Ilustración 18 Ejemplo de reproducción de partida mediante GameDirective 50
Ilustración 19 Ejemplo de controles externos en GameDirective 51
Ilustración 20 Aspecto de la directiva BattleCreatorDirective con una configuración lista para enviar 51
Ilustración 21 Aspecto de la directiva BattleCreatorDirective durante la ejecucón de una batalla 51
Ilustración 22 Ejemplo de instancias de la directiva BattleCreatorButtonDirective 52
Ilustración 23 Ejemplo de instancia de la directiva ProfileUpdateDialogDrtv 53
Ilustración 24 Ejemplo de instancia de la directiva TournamentJoinDrtv 53
Ilustración 25 Aspecto de la interfaz 'Listado de batallas' 56
Ilustración 26 Detalle de autor del agente en 'Listado de batallas' 56
Ilustración 27 Detalle de hora de batalla en 'Listado de batallas' 56
Ilustración 28 Aspecto de la interfaz 'Visualización de batalla' 57
Ilustración 29 Detalle del reproductor en la interfaz 'Visualización de batalla' 58
Ilustración 30 Detalle de los controles en la interfaz 'Visualización de batalla' 58
Ilustración 31 Detalle del estado de equipo en la interfaz 'Visualización de batalla' 58
Ilustración 32 Aspecto de la interfaz 'Creación de la batalla' 59
Ilustración 33 Detalle creación de batalla incompleta en la interfaz 'Creación de la batalla' 59
Ilustración 34 Aspecto de la interfaz 'Listado de torneos' 60
Ilustración 35 Detalle del botón para unirse al torneo en interfaz 'Listado de torneos' 61
Ilustración 36 Detalle hora de creación en interfaz 'Listado de torneos' 61
Ilustración 37 Aspecto de la interfaz 'Listado de torneos' 61
Ilustración 38 Aspecto de la interfaz 'Detalle de torneo' 62

Ilustración 39 Detalle de información de torneo en la interfaz 'Detalle de torneo' 62
Ilustración 40 Detalle de configuración de rondas en la interfaz 'Detalle de torneo'..... 63
Ilustración 41 Detalle de indicador de estado de torneo en la interfaz 'Detalle de torneo' 63
Ilustración 42 Detalle de rondas en la interfaz 'Detalle de torneo'..... 63
Ilustración 43 Aspecto de la interfaz 'Listado de usuarios'..... 64
Ilustración 44 Aspecto de la interfaz 'Detalle de usuario' 65
Ilustración 45 Aspecto de la interfaz 'Crear nuevo agente'..... 66
Ilustración 46 Aspecto de la interfaz 'Editar Agente'..... 67
Ilustración 47 Detalle de hora en el listado de versiones en la interfaz 'Editar agente' 67
Ilustración 48 Diagrama de flujo de particionado 70
Ilustración 49 Arquitectura del Servidor..... 75
Ilustración 50 Esquema de base de datos del subsistema servidor 79
Ilustración 51 Ejemplo de flujo de eventos y callbacks 84
Ilustración 52 Ejemplo de flujo de multiples callbacks sin promesas..... 86
Ilustración 53 Ejemplo de flujo de multiples callbacks con promesas 88
Ilustración 54 Detalle botón para actualizar agente 104
Ilustración 55 Detalle botón acceso a perfil de usuario 104

1 INTRODUCCIÓN

Una vez finalizada la carrera, llega la recta final: El trabajo de fin de grado.

La universidad cuenta con un listado de propuestas de libre elección, pero tanto a mí como a mi compañero y gran amigo *Luis Sebastián Huerta* no nos llaman la atención lo suficiente. Es entonces cuando empezamos a pensar que queremos algo nuestro; algo que cuando alguien nos pregunte "¿Y de que estás haciendo el proyecto?", sonriéramos y nos sintiéramos orgullosos de explicar *dónde nos habíamos metido*. Es en este punto donde surge la idea de llevar a cabo JSWARS.

Durante los últimos años, hemos visto como la inteligencia artificial dejaba de ser algo desconocido y oscuro para el gran público, para pasar a ser una disciplina *cercana, atrayente*, y con un *potencial* muy grande.

Personalmente, no puedo sacarme de la cabeza *Mario AI Championship*, con su pequeño mario rebotando a toda velocidad de formas imposibles; o *AI Birds,* la implementación de *Angry Birds* controlada por Inteligencia Artificial que realiza disparos casi perfectos.

La fascinación por la Inteligencia artificial es algo que ambos compartíamos, por lo que lo vimos claro, y una vez encontráramos a alguien que nos ayudara a enfocarlo como *Trabajo de fin de grado*, es resto era *echar horas.*

Una vez con *Carlos Linares* como tutor*, Luis Sebastián* a la cabeza de la parte del *motor de juego* y *agentes inteligentes*, *y yo* a la cabeza del *cliente* y *servidor,* hemos conseguido superar el reto que representaba para nosotros JSWARS.

En las siguientes páginas encontrarás información del análisis y desarrollo de la parte web (*cliente* y *servidor*).

La parte de implementación del motor y los agentes inteligentes debe consultarse en el TFG: ***Investigación y desarrollo de una máquina virtual que permita la ejecución de agentes inteligentes en JavaScript****. (Sebastián Huerta, 2016)*

2 MOTIVACIÓN Y OBJETIVOS

A día de hoy, y desde que surgió, existe una gran comunidad de entusiastas de la inteligencia artificial. Desde su aparición como objeto de estudio alrededor de los años 50 (Wikipedia, Inteligencia artificial, s.f.), la inteligencia artificial ha evolucionado de manera exponencial hasta nuestros días, llegando durante este tiempo a encontrar inteligencias *perfectas* que son capaces de resolver problemas consiguiendo un desarrollo perfecto de la solución. Por suerte para todos aquellos amantes de esta rama de la computación, en la gran mayoría de casos aún hay mucho por hacer.

2.1 MOTIVACIÓN

Con ánimo de poner a prueba las habilidades de los más entusiastas, desde hace unos años vienen surgiendo competiciones de inteligencia artificial. Estas competiciones parten de un planteamiento sencillo: Un problema (Normalmente en formato juego) a resolver en el que los distintos jugadores deben competir usando inteligencia artificial, en lugar de hacerlo usando su propia inteligencia. La manera en el que compiten las distintas inteligencias varía según el problema planteado en el juego, pero en rasgos generales podemos hablar de:

- **Competiciones 'single player':** Una única inteligencia artificial ejecuta a la vez para intentar dar solución al problema planteado con el menor número de recursos posibles. El recurso a medir para determinar el ganador lo define el juego, por lo que puede ser tiempo, o cualquier otro parámetro. *Las inteligencias ejecutan en serie.*
- **Competiciones 'multi player':** Varias inteligencias artificiales ejecutan a la vez para resolver el mismo problema. En este caso el problema a solucionar suele ser eliminar a la otra inteligencia artificial utilizando las herramientas accesibles dentro del juego, quedando ganadora aquella que consiga sobrevivir más tiempo. *Las inteligencias ejecutan en paralelo.*

La motivación principal de este proyecto es atraer, mediante competiciones de IA tipo *multi player*, tanto a los entusiastas de la inteligencia artificial como a aquellos que nunca han tenido oportunidad de conocer este interesante mundo. La manera de acercarlo a estos usuarios menos expertos será facilitando al máximo el acceso a la plataforma (Mediante el login en GitHub, que no requiere registro) y mostrando los resultados de las batallas en tiempo real, resultando más visual y atrayente, todo englobado en un ambiente altamente competitivo.

2.2 OBJETIVOS Y ALCANCE DEL PROYECTO

El objetivo principal es la creación de una plataforma que permita a los usuarios la creación y evaluación de agentes (Pequeños programas) inteligentes escritos en JavaScript que controlarán un conjunto reducido de unidades. Estas unidades tienen la capacidad de moverse y disparar, con el fin de derrotar a todas las unidades controladas por el agente enemigo. Los agentes se pueden enfrentar en batallas sencillas uno contra uno o en el contexto de torneos, en los que tras varias batallas, y según los resultados de las mismas, se obtiene un agente ganador. El cliente, el servidor y los agentes correrán todos sobre JavaScript.

El alcance de este proyecto se limita al diseño y desarrollo de la plataforma web que permite crear agentes, ordenar ejecución de batallas, gestionar las batallas y el conjunto de usuarios. La ejecución de las partidas una vez que son solicitadas *se ha desarrollado en paralelo en otro trabajo de fin de grado independiente:* **Investigación y desarrollo de una máquina virtual que permita la ejecución de agentes inteligentes en JavaScript**. (Sebastián Huerta, 2016)

Partiendo del alcance del proyecto definido, se definen los siguientes objetivos:

- Diseño e implementación de un **cliente web** usando Angular JS que exponga a los usuarios finales todas las operaciones disponibles en la API (*Inicio de sesión*, *creación y modificación de agentes*, *solicitud de batallas*, etc…).
- El cliente contará con un *reproductor de batallas* que permitirá reproducirlas una vez se han ejecutado. Este, además permitirá avanzar y retroceder dentro del tiempo de ejecución de la batalla.
- Diseño e implementación de una *API Rest* en JSON que facilite la comunicación del cliente con el servidor y exponga todas las operaciones disponibles en la plataforma.
- Diseño e implementación de un **servidor web en Node JS** que se ocupe de gestionar toda la información sobre los *usuarios*, *agentes*, *batallas* y *campeonatos*. Este servidor web contará con una **base de datos Mongo DB** que también funciona sobre JavaScript
- Integrar **GitHub como método único de inicio de sesión**, evitando a los usuarios la creación de una cuenta en nuestra plataforma.
- Diseño e implementación de un **sistema de ejecución de batallas asíncrono** sobre NodeJS que permita *ejecutar las batallas* dentro del servidor sin bloquear la ejecución del mismo y de manera desatendida.

3 ESTADO DE LA CUESTIÓN

La inteligencia artificial tiene un campo muy amplio de aplicaciones. En la actualidad podemos encontrar aplicaciones que van desde crear aviones sin piloto (International Aerial Robotics Competition) y coches sin conductor (DARPA Grand Challenge) hasta jugar partidas de ajedrez casi perfectas (World Computer Chess Championship), pero lo cierto es que en la gran mayoría de ellos hay un factor común: Los grandes avances y la mayor repercusión en todas las aplicaciones se consiguen en el contexto de competiciones.

Si bien la parte de inteligencia artificial sobre la que se apoya el proyecto es algo más humilde, comparte de igual forma la capacidad de potenciación mediante la competitividad. Y es en ese punto exactamente donde este proyecto toma todo su sentido: Convertir un simple problema de inteligencia artificial en algo divertido, social y que fomente la competitividad.

El diseño de sistemas que acerquen la inteligencia artificial a las personas mediante el formato competición no es algo nuevo, pero es después de la evolución de las *Web 2.0* cuando se empiezan a ver grandes avances en el tema, gracias a las nuevas tecnologías web.

El acercamiento clásico de las competiciones de inteligencia artificial era mediante la evaluación en diferido: Un participante trabajaba en su máquina local con un software convenientemente preparado para realizar entrenamientos y afinar su programa. Una vez que el participante lo consideraba terminado, procedía a proporcionarle el código al organizador (mediante cualquier vía disponible), y este ejecutaba las partidas en cualquier momento publicando posteriormente el resultado final. Los problemas de formato son:

- La capacidad para generar y participar en desafíos es limitada, puesto que tanto la creación como la participación en ellos está supeditada a la acción de un organizador.
- La competitividad entre personas directas no es posible.
- Se produce una pérdida de atención en la plataforma por parte del usuario debido a los tiempos de espera para la ejecución de los torneos.
- No se dispone de movilidad alguna a la hora de hacer el desarrollo, puesto que se requiere software específico en cada ordenador.

En las siguientes líneas, vamos a estudiar todos los sistemas que se han tenido en cuenta. Entre ellos podemos ver sistemas competitivos de inteligencia artificial que se apoyan en una arquitectura web, y otros relacionados con otras ramas de la computación pero que cuenten con usabilidad parecida (Sobre todo aquellos que permiten editar y subir código en línea dentro de una plataforma para su posterior evaluación). Todos ellos representan el *estado del arte* de los sistemas competitivos online mediante publicación de código en entornos web, que es donde se enmarcaría nuestro proyecto.

3.1 COMPETICIONES DE IA

3.1.1 Elevator Saga

Disponible en: http://play.elevatorsaga.com/

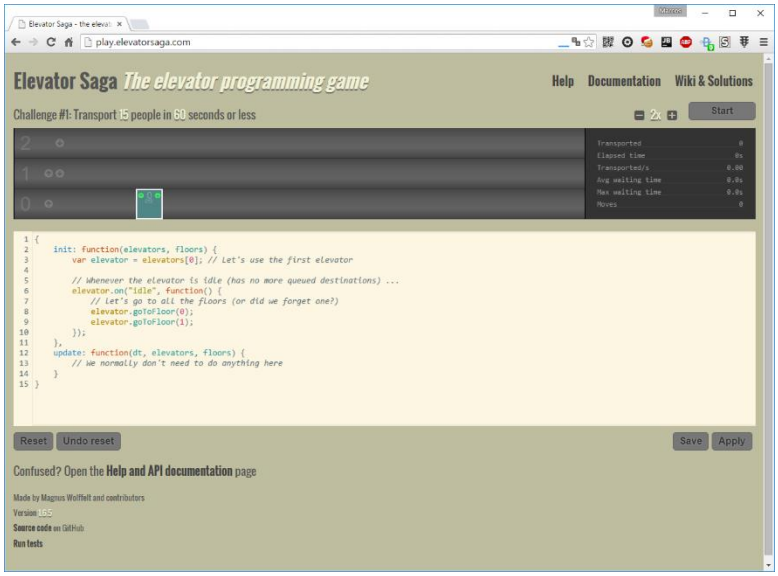


Ilustración 1 *Aspecto de Elevator Saga*

Elevator saga fue creado por Magnus Wolffelts y algunos colaboradores anónimos hicieron también sus aportes a la plataforma a través de pull request en GitHub. El principio del juego es sencillo: hay que desarrollar un algoritmo en JavaScript que controle de la manera más eficiente posible un simulador de un ascensor real. Para desarrollar el algoritmo, el sistema deja disponible al programador una API perfectamente definida, que este puede usar según sus necesidades. Podríamos considerarlo del tipo *single player.*

Sus puntos positivos son:

- API de IA bien estructurada, correctamente documentada y en la que se incluyen ejemplos.
- Ejecución del algoritmo en el momento (No requiere espera).
- Editor con sintaxis resaltada.

Sus puntos negativos son:

- No existe ninguna manera directa de comparar los resultados de dos algoritmos. Toda la ejecución se hace en cliente y no se comparte en ningún servidor.
- La plataforma no cuenta con un servidor que almacene los algoritmos creados por los usuarios ni los resultados de las ejecuciones. Lo usuarios no puede recuperar los códigos anteriores.
- Sistema con funcionalidad extremadamente reducida.

3.1.2 Ants AI Challenge

Disponible en: http://ants.aichallenge.org/

Ants AI Challenge es una competición *multi player* en el que la inteligencia artificial de cada equipo debe trabajar para destruir al resto de hormigas de equipos enemigos. Aunque la reproducción de las partidas se realiza en una arquitectura web, la programación de la inteligencia artificial se realiza en local y la ejecución se hace en diferido en el servidor.

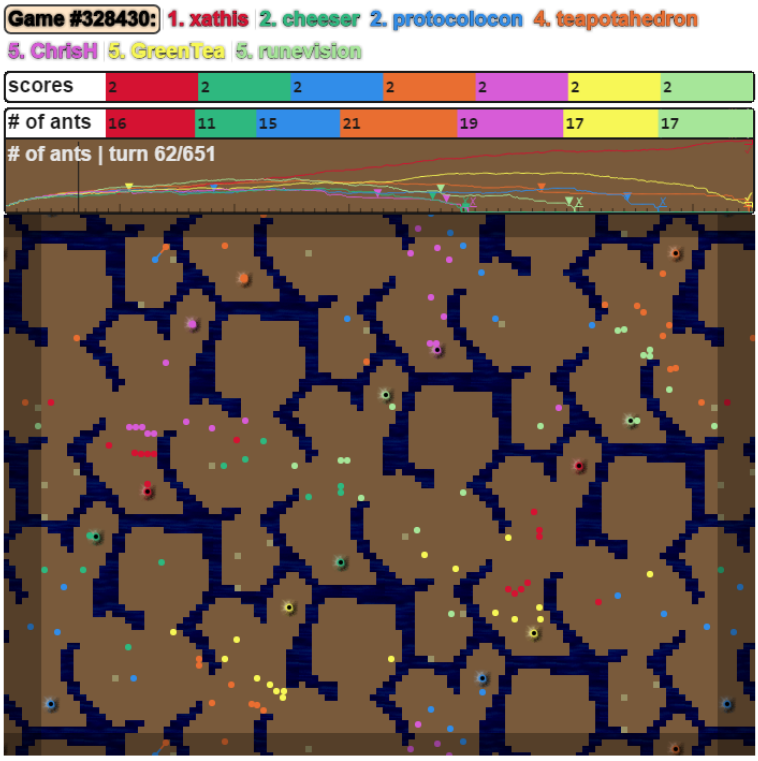


Ilustración 2 *Aspecto de Ants AI Challenge*

En este caso el servidor permite él envió del código del agente inteligente en una gran cantidad de lenguajes, aunque este puede tardar hasta una hora en ejecutar y ser visible. Además luchará contra contrincantes desconocidos (No existe posibilidad de luchar contra alguien en concreto).

Sus puntos positivos son:

- Diseño de mapa minimalista y claro. Identificación de los equipos por colores.
- Permite seleccionar país de origen a los usuarios. Después se puede usar para rankings.
- Soporte para participación mediante el uso de múltiples lenguajes de programación.

Sus puntos negativos son:

- Tiene un proceso registro (No se permite iniciar sesión con servicios de terceros). Además hay que activar la cuenta mediante una confirmación por correo.
- Es necesario descargar un software específico para desarrollar y probar la inteligencia artificial a desarrollar en local.
- Las batallas no se ejecutan en tiempo real, es necesario subir el código y esperar un plazo de hasta 1 hora.
- No es posible luchar contra usuarios concretos.

3.2 OTRAS COMPETICIONES

3.2.1 CodeFights

Disponible en: https://codefights.com/home

CodeFights es una plataforma para resolver problemas de programación. Se ofrece un código erróneo y se pide corregirlo para que funcione correctamente. Gana el usuario que consiga este objetivo antes. Aunque el sistema no se apoya sobre inteligencia artificial, el tipo de interfaces con el que cuentan son muy parecidas en funcionalidad a la que se necesita en nuestro sistema.

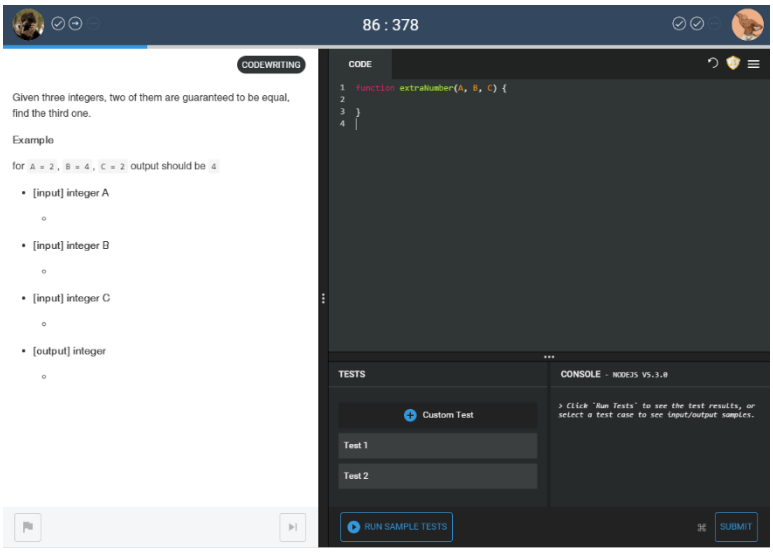


Ilustración 3 *Aspecto de CodeFight*

Todas las interfaces son muy agradables y la experiencia de usuario es espectacular (Hace uso de Google Material, un framework de diseño creado por Google), por lo que usaremos como guía para mejorar el diseño, sobre todo en líneas futuras.

Sus puntos positivos son:

- Experiencia de usuario muy agradable; interfaces cuidadas al milímetro.
- Inicio de sesión mediante múltiples plataformas externas (Google, Facebook, Github y Twitter)
- Cuenta con el concepto de torneos y además con logros.

Sus puntos negativos son:

- No tiene nada de inteligencia artificial ni de reproducción de partidas, por lo que en ese aspecto no nos sirve de inspiración.

3.2.2 HackerEarth

Disponible en: https://www.hackerearth.com/

HackerEarth es una plataforma que ofrece retos a resolver mediante programación. Los retos pueden tener como restricción cierto lenguaje, o ser de resolución libre con cualquiera de los lenguajes de programación ofrecidos. Está enfocado tanto a empresas que buscan contratar personal cualificado como al entretenimiento en general. Todo se apoya sobre el concepto de reto. Es interesante para este proyecto porque comparte el uso de editores y gestión de código en entornos web.

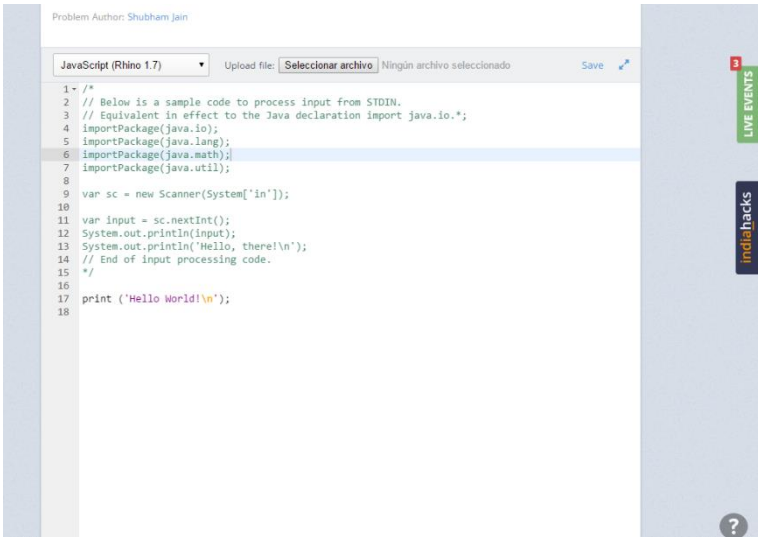


Ilustración 4 *Aspecto de HackerEarth*

Sus puntos positivos son:

- Gamifica el proceso de completado del perfil, animando al usuario a aportar más datos sobre su persona.
- Inicio de sesión mediante múltiples plataformas externas (Google, Facebook, Github y Twitter)
- Se puede seguir a otros usuarios, viendo sus actualizaciones cuando se producen. Esta parte la integraremos en nuestras líneas futuras.
- Se ha sabido explotar económicamente la plataforma con la inclusión de empresas como creadoras de retos.
- Incluye algunos retos de inteligencia artificial.

Sus puntos negativos son:

- Diseño demasiado complejo. Las páginas de listados de retos están llenas de datos y de imágenes, lo cual vuelve su aspecto algo caótico y recargado.

4 DESARROLLO

En este punto se van a exponer todos los detalles inherentes al proceso de análisis y diseño del conjunto del sistema, entendido como la suma de los siguientes subsistemas:

- **Subsistema cliente:** Es la parte del sistema que ejecuta en el explorador del cliente. Tiene la responsabilidad de mostrar todas las interfaces contra las que el usuario interactúa, realizando las operaciones contra el subsistema servidor.
- **Subsistema servidor:** Es la parte del sistema que ejecuta en el servidor. Tiene como responsabilidad el manejo y control de toda la lógica de negocio, desde la *API Rest* hasta las tablas en la base de datos, pasando por la ejecución de batallas y torneos. También se ocupa de servir al usuario final el código del subsistema cliente.

El análisis y diseño de cada uno de los subsistemas por separado puede ser encontrado en los puntos Subsistema cliente y Subsistema servidor.

4.1 ANÁLISIS

En este punto estudiaremos los objetos presentes en el conjunto del sistema; veremos su conceptualización y expondremos aquellos detalles que son más importantes para la comprensión de cada uno. Posteriormente generaremos unos requisitos y casos de uso específicos alrededor de cada uno.

4.1.1 Objetos del sistema

En este punto se conceptualizará cada uno de los objetos del sistema, intentando comprender a que deben su existencia, que aportan al conjunto, y cuáles son sus principales características, limitaciones y relaciones con otros objetos.

4.1.1.1 Usuario

El usuario es el objeto básico del sistema que permite a las personas interactuar con la plataforma participando en batallas y torneos (los usuarios anónimos no pueden). En la inmensa mayoría de aplicaciones, la base de datos contiene una tabla ‘usuarios’ con al menos un campo ‘usuario’ y otro ‘contraseña’. Esos datos se usan para identificar al usuario cuando desea iniciar la sesión y así poder personalizar su experiencia. En este proyecto se ha optado por un inicio de sesión apoyado en GitHub. La gran ventaja de este sistema es que evita tener que desarrollar un registro integrado (GitHub ya cuenta con uno muy fiable), y una vez iniciada la sesión, la experiencia de usuario es igual la de un sistema clásico. Respecto a las desventajas, una de las más grandes es el hecho en sí de que es obligatorio tener cuenta en GitHub, aunque el tipo de personas que pueden estar interesadas en la plataforma muy probablemente ya la tendrán.

4.1.1.2 Agente

El agente es el objeto que permite a un usuario participar en las batallas. Una vez que el usuario se ha registrado, tiene acceso a ver y modificar todos sus agentes. El agente no es más que el *código fuente* de una inteligencia artificial creada por un usuario. El código de los agentes (sus modificaciones) es auditado, permitiendo la recuperación por su dueño de versiones anteriores y relacionar una batalla con una *versión* concreta de un *agente*. Si no se hiciera de esta forma, algunas batallas aparecerían relacionadas con códigos que no serían con el que ejecutaron en su momento.

4.1.1.3 Batalla

La batalla es el objeto que permite competir a los agentes. Aunque el servidor tiene preparación para otras configuraciones con más agentes, el tope máximo y mínimo actual de participación es de 2 agentes. Las batallas pueden ser reproducidas tanto por usuarios anónimos como con sesión iniciada, pero la solicitud de ejecución de estas tiene que ser mediante un usuario con sesión iniciada. La ejecución se realiza inmediatamente tras su solicitud, y se notifica al usuario cuando está lista, ofreciendo además en ese momento acceso a un volcado de toda la partida en formato JSON para su análisis.

4.1.1.4 Mapa

El mapa es el objeto que contiene la información del entorno sobre el que se pueden mover los agentes durante su ejecución en el contexto de una batalla. Aunque el sistema está preparado para ello, se ha limitado la capacidad de agregar mapas y actualmente se selecciona siempre el mapa por defecto para la ejecución de las partidas. El formato esperado de los mapas es el que genera el software **Tiled**. El aspecto de los mapas puede ser texturizado o plano.

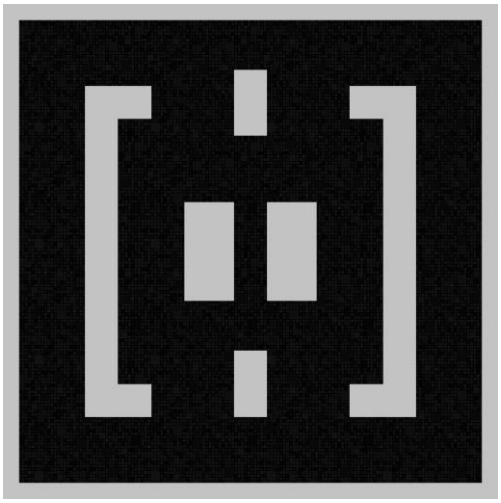


Ilustración 5 *Ejemplo de mapa y su posible aspecto*

4.1.1.5 Torneo

El torneo es el objeto que permite realizar competiciones complejas entre agentes, utilizando para ello batallas clásicas entre parejas de agentes. A la hora de crear el torneo se especifica el número de rondas que este tendrá. Son las rondas las que determinan la capacidad máxima de agentes que deben apuntarse al torneo para que tenga lugar. La fórmula para calcular el total de agentes que pueden participar en un torneo es $Participantes = 2^{Nº\ Rondas}$ (En un torneo con 4 rondas, juegan 16 agentes).

En el momento que una batalla llega a su máximo de inscripciones, esta se ejecuta. La preparación y ejecución de las rondas se hace en orden inverso (Primero la ronda 4, luego la 3 y así hasta la 1) siendo la ronda 4 la que más batallas tiene y la 1 la final. Para más información consultar el punto **Cola de ejecución de torneos (**TournamentRunner**)**. El esquema de ejecución de batallas es el siguiente:

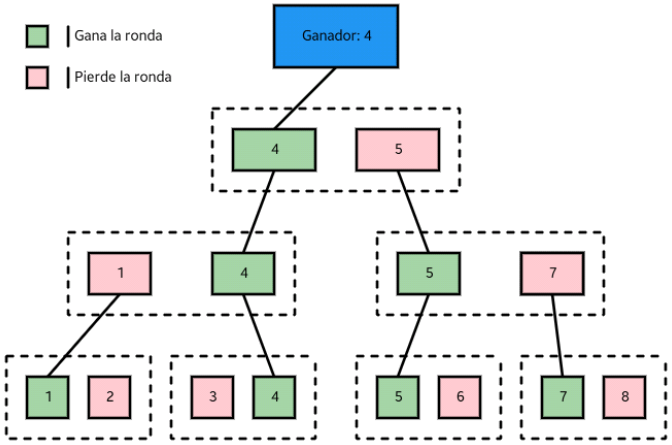


Ilustración 6 *Formato torneos*

4.1.2 Requisitos del sistema

En este punto se van a exponer los requisitos que debe tener el sistema final, una vez terminado. Para mayor claridad, se han organizado los requisitos por apartados en función al concepto sobre el que deben su existencia. El formato seguido para la definición de los requisitos es el siguiente:

- **Identificador:** identificador del requisitos. Debe tener el formato **R-[Inicial]-[Número]** donde:
  - *Inicial***:** es el código de objeto (Permite diferencias lo requisitos por objetos a simple vista)
  - *Número***:** Número incremental del requisito. Los requisitos de cada objeto tienen su propia numeración.
- **Prioridad:** Puede ser Alta, Media o Baja. Sirve para colocar unos requisitos por delante de otros en importancia

- **Tipo de Requisito:** Puede ser *funcional* (Define algo que el sistema debe permitir*) o no funcional* (define algo que el sistema no debe permitir).
- **Descripción:** Explicación detallada y lo más sencilla posible del requisitos.
- **Subsistema:** Puede ser cliente o servidor

Quedando cada tabla de la siguiente forma:

Identificador	R-[*Inicial*]-[*Número*]:		
Prioridad	[*Alta* | *Media* | *Baja*]	Tipo de requisito	[*Func.* | *No Func.*]
Descripción	[*Descripción*]		
Subsistema	[*Cliente* | *Servidor*]		

4.1.2.1 Globales

Identificador	R-G-001		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema tiene que estar basado en JavaScript.		
Subsistema	Cliente y Servidor		

Identificador	R-G-002		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe iniciar la sesión de usuario usando la sesión de Github en lugar de con un password.		
Subsistema	Servidor		

4.1.2.2 Usuario

Identificador	R-U-001		
Prioridad	Alta	Tipo de requisito	Funcional

Descripción	El sistema debe iniciar la sesión del usuario desde Github cuando se solicite
Subsistema	Cliente y Servidor

Identificador	R-U-002		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe actualizar la información del perfil usando la de Github cada vez que un usuario inicie la sesión desde Github		
Subsistema	Servidor		

Identificador	R-U-003		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir modificar la información del perfil al usuario		
Subsistema	Cliente y Servidor		

Identificador	R-U-004		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir cerrar la sesión al usuario (Desvincularlo de sus sesión de Github)		
Subsistema	Cliente y Servidor		

Identificador	R-U-004		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe ofrecer un listado público y paginado de los usuarios registrados		
Subsistema	Cliente y Servidor		

Identificador	R-U-005		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe restringir el acceso público a solo los siguiente datos del usuario: Nombre de usuario, Identificador y Avatar		
Subsistema	Servidor		

4.1.2.3 Agente

Identificador	R-A-001		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe ofrecer la posibilidad de crear agentes al usuario		
Subsistema	Cliente y servidor		

Identificador	R-A-002		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe ofrecer un histórico de versiones de todos los agentes de la plataforma.		
Subsistema	Cliente y servidor		

Identificador	R-A-002		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe ofrecer la posibilidad de recuperar un agente y toda su información.		
Subsistema	Cliente y servidor		

Identificador	R-A-003

Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe ofrecer la posibilidad de obtener un listado público de los agentes de un usuario.		
Subsistema	Cliente y servidor		

Identificador	R-A-004		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe restringir el acceso público a solo los siguientes datos de agente: Fecha de creación, nombre, victorias y pérdidas. El resto de datos solo están disponibles si el usuario es dueño del agente solicitado.		
Subsistema	Cliente y servidor		

4.1.2.4 Batalla

Identificador	R-B-001		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir el acceso a las batallas almacenadas (Tanto su información básica como cada uno de sus *fotogramas.*		
Subsistema	Cliente y servidor		

Identificador	R-B-002		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir listar y paginar las batallas ya ejecutadas		
Subsistema	Cliente y servidor		

Identificador	R-B-003

Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir solicitar batallas entre dos agentes		
Subsistema	Cliente y servidor		

Identificador	R-B-004		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe limitar las solicitudes de batalla a usuarios con sesión iniciada (Los usuarios anónimos no pueden solicitar una batalla)		
Subsistema	Servidor		

Identificador	R-B-005		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe ejecutar las batallas en un hilo de proceso diferente para no paralizar las peticiones web cada vez que se está ejecutando una partida. (JavaScript solo tiene un hilo por defecto)		
Subsistema	Servidor		

Identificador	R-B-006		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe almacenar la ejecución de las batallas para su posterior reproducción.		
Subsistema	Servidor		

Identificador	R-B-007		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe limitar el tiempo máximo de partida para evitar partidas sin fin en caso de agentes inertes o demasiado inexpertos.		

Subsistema	Servidor

Identificador	R-B-008		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe distinguir durante la reproducción de una batalla a cada uno de los agentes mediante colores.		
Subsistema	Cliente y servidor		

Identificador	R-B-009		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir avanzar y retroceder durante la reproducción de una partida desde el cliente.		
Subsistema	Cliente y servidor		

Identificador	R-B-010		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe informar del progreso de la batalla, incluyendo el estado de las unidades de cada agente.		
Subsistema	Cliente y servidor		

Identificador	R-B-011		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	Durante la reproducción, el sistema debe representar sobre el mapa tanto la posición de cada agente como la de cada bala presente.		
Subsistema	Cliente y servidor		

Identificador	R-B-012		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe ofrecer la posibilidad de descargar un volcado de la ejecución de la partida al usuario final con el fin de permitirle usarlo para estudiarlo y desarrollar otros agentes en función a los resultados.		
Subsistema	Cliente y servidor		

Identificador	R-B-013		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe avisar en tiempo real al usuario que solicitó la batalla cuando esta se encuentra disponible para su reproducción.		
Subsistema	Cliente y servidor		

4.1.2.5 Mapa

Identificador	R-M-001		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe ser capaz de interpretar aquellos mapas diseñados con la herramienta de diseño de mapas Tiled, sin requerir procesado adicional de este.		
Subsistema	Servidor		

Identificador	R-M-002		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe tener la capacidad de almacenar más de un mapa para poder usarlos en las batallas.		
Subsistema	Cliente y servidor		

Identificador	R-M-003		
Prioridad	Alta	Tipo de requisito	No funcional
Descripción	El sistema debe contar con un mapa por defecto		
Subsistema	Servidor		

4.1.2.6 Torneo

Identificador	R-T-001		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe permitir torneos		
Subsistema	Servidor		

Identificador	R-T-002		
Prioridad	Alta	Tipo de requisito	No Funcional
Descripción	El sistema debe limitar la creación de torneos a usuarios administradores		
Subsistema	Servidor		

Identificador	R-T-003		
Prioridad	Alta	Tipo de requisito	Funciona
Descripción	El sistema debe permitir listar los torneos según su estado. (Ejecutado, Pendiente, Cancelado)		
Subsistema	Cliente y servidor		

Identificador	R-T-004		
Prioridad	Alta	Tipo de requisito	Funcional

Descripción	El sistema debe permitir a un usuario suscribirse a un torneo
Subsistema	Cliente y servidor

Identificador	R-T-005		
Prioridad	Alta	Tipo de requisito	No Funcional
Descripción	El sistema debe limitar la suscripción a los torneos a un máximo de una inscripción por usuarios y torneo.		
Subsistema	Servidor		

Identificador	R-T-006		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe almacenar los resultados del torneo así como cada una de las batallas relacionadas con la ejecución de ese torneo		
Subsistema	Servidor		

4.1.2.7 Diseño

Identificador	R-DI-001		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe poder visualizarse correctamente en exploradores móviles.		
Subsistema	Cliente		

Identificador	R-DI-002		
Prioridad	Alta	Tipo de requisito	Funcional

Descripción	El sistema debe estar compuesto por un único documento que cambie su contenido mediante peticiones asíncronas.
Subsistema	Cliente

4.1.2.8 Documentación

Identificador	R-DO-001		
Prioridad	Alta	Tipo de requisito	Funcional
Descripción	El sistema debe contar con una sección en el que se explique claramente cuál es la API con la que se pueden programar los agentes, concretamente la manera de acceder a la información y como se deben realizar las llamadas.		
Subsistema	Cliente		

4.1.3 Casos de Uso

En este punto se analizarán las acciones que un usuario podrá realizar en la plataforma, en función del rol que desempeñe en ese momento dentro de la plataforma. Los roles o actores existentes son:

- **Usuario anónimo:** Son todos aquellos usuarios que no han iniciado la sesión en la plataforma todavía. Este es el rol de todo usuario que accede a la plataforma por primera vez. También puede tomar este rol un usuario que pierde su sesión al cerrarla de manera consciente o al limpiar las cookies.
- **Usuario con sesión:** Son todos aquellos usuarios que han conseguido una sesión después de hacer un inicio de sesión correcto a través de GitHub.
- **Usuario administrador:** Son todas aquellas personas que tienen privilegios superiores al resto y deben su existencia a la necesidad de poder hacer ciertas tareas de mantenimiento dentro del sistema. Este rol no está presente en la plataforma pública, si no que se accede a él desde la capa de sistemas del servidor.

El formato seguido para la definición de los requisitos será el siguiente.

- **Identificador:** Identificador del requisito. Debe tener el formato **R-[Inicial]-[Número]** donde:
  - *Inicial***:** Es el código de objeto (Permite diferencias lo casos de uso por objetos a simple vista)
  - *Número***:** Número incremental del caso de uso. Los casos de uso de cada objeto tienen su propia numeración.
- **Actores:** Roles o actores que participan en el caso de uso.
- **Descripción:** Explicación detallada y lo más sencilla posible del caso de uso.
- **Precondiciones:** Condiciones necesarias para que el caso de uso pueda aplicarse.
- **Pasos a seguir:** Pasos que debe seguir el actor para conseguir completar el caso de uso

Quedando cada tabla de la siguiente forma:

Caso de uso	[*Caso de uso*]	Identificador	R-[*Inicial*]-[*Número*]:
Actores	[*Actores involucrados*]		
Descripción	[*Descripción*]		
Precondiciones	[*Cliente	Servidor*]		
Pasos a seguir	[*Escenario*]		

Para mayor claridad, se han organizado los casos de uso por apartados en función al concepto sobre el que deben su existencia.

4.1.3.1 Usuario

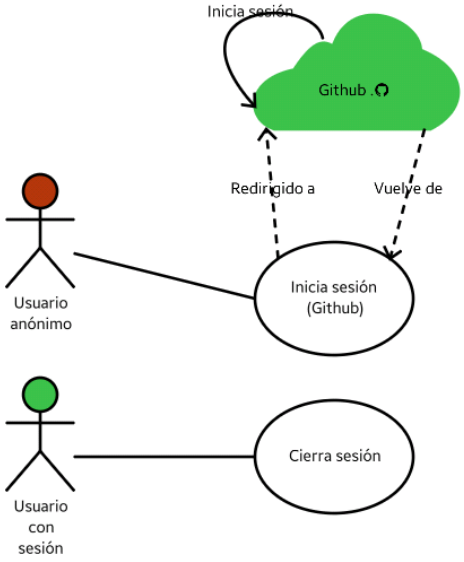


Ilustración 8 *Diagrama de casos de uso en referencia al inicio de sesión del objeto usuario*

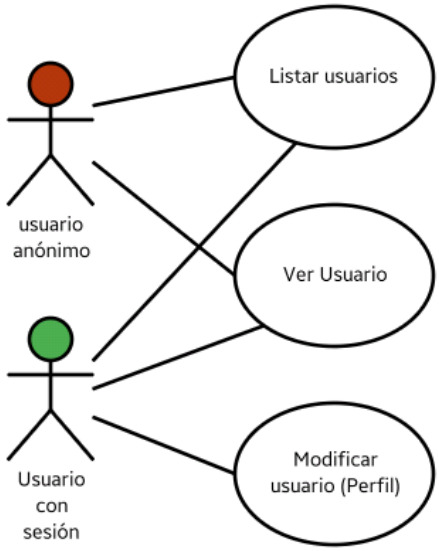


Ilustración 7 *Diagrama de casos de uso en referencia al objeto Usuario*

Caso de uso	Iniciar sesión	Identificador	R-U-001
Actores	Usuario anónimo		
Descripción	Conseguir una sesión de usuario válida para poder interactuar con la plataforma.		
Precondiciones	Usuario anónimo		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar en el botón “Sign in with GitHub” del menú. 3. Si no hay sesión previamente iniciada en GitHub, introducir los datos y enviar el formulario.		

Caso de uso	Cerrar sesión	Identificador	R-U-002
Actores	Usuario con sesión		
Descripción	Salir de la sesión de la que el usuario dispone actualmente		
Precondiciones	Usuario con sesión iniciada		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar en el botón “Logout” del menú.		

Caso de uso	Listar usuarios	Identificador	R-U-003
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Obtener una lista con los usuarios registrados en la plataforma		
Precondiciones	Ninguna		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar en el botón “Users” del menú.		

Caso de uso	Ver usuario (Información)	Identificador	R-U-004
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Obtener relativa a un usuario de la plataforma.		
Precondiciones	Ninguna		
Pasos a seguir	1. Acceder al sitio web. 2. Navegar a cualquier sección con enlaces a la información del perfil de un usuario (No tiene por qué ser en “Users”). 3. Pulsa en el nombre de usuario del que se quiere obtener la información.		

Caso de uso	Modificar usuario (Información)	Identificador	R-U-005
Actores	Usuario con sesión		
Descripción	Modificar la información de un usuario.		
Precondiciones	Persona con sesión iniciada con el mismo usuario que intenta modificar.		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. En el perfil, pulsar en el icono del lapicero junto al nombre. 4. Rellenar los campos del formulario que aparece y pulsar “Update”.		

4.1.3.2 Agente

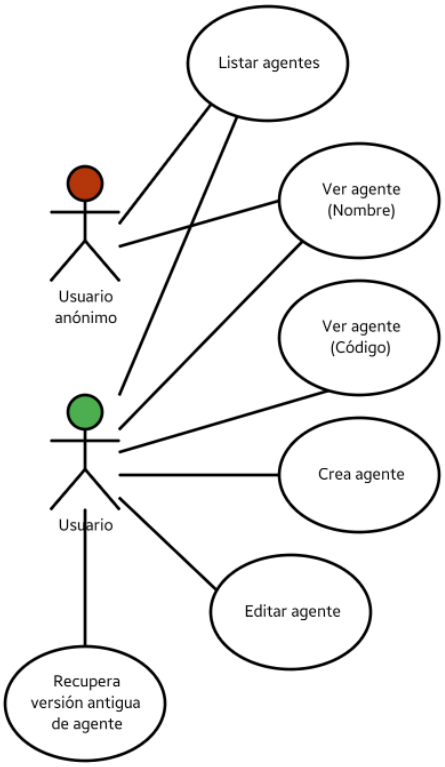


Ilustración 9 *Diagrama de casos de uso en referencia al objeto Agente*

Caso de uso	Listar agentes	Identificador	R-A-001
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Obtener una lista con los agentes registrados para cierto usuario		
Precondiciones	Ninguna		
Pasos a seguir	1. Acceder al sitio web. 2. Entrar al perfil de cualquier usuario registrado en la plataforma.		

Caso de uso	Ver agente (Código)	Identificador	R-A-002
Actores	Usuario con sesión iniciada		
Descripción	Obtener el código de un agente concreto.		
Precondiciones	El usuario que intenta ver el código del agente debe ser el propietario del agente		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. En el listado de agentes, pulsar sobre el nombre de uno de ellos.		

Caso de uso	Ver agente (Código)	Identificador	R-A-003
Actores	Usuario con sesión iniciada		
Descripción	Obtener el código de un agente concreto.		
Precondiciones	El usuario que intenta ver el código del agente debe ser el propietario del agente		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. En el listado de agentes, pulsar sobre el nombre de uno de ellos.		

Caso de uso	Crear agente	Identificador	R-A-004
Actores	Usuario con sesión iniciada		
Descripción	Crear un nuevo agente con su código asociado		
Precondiciones	Ninguna		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. Pulsar en “Create new agent”		

Caso de uso	Editar agente	Identificador	R-A-005
Actores	Usuario con sesión iniciada		
Descripción	Crear un nuevo agente con su código asociado		
Precondiciones	Usuario con sesión iniciada y al menos con un agente registrado.		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. Pulsar en “Create new agent”.		

Caso de uso	Recuperar versión antigua agente	Identificador	R-A-006
Actores	Usuario con sesión iniciada		
Descripción	Obtener una versión antigua del código de un agente.		
Precondiciones	Usuario con sesión iniciada y al menos con un agente registrado con alguna versión antigua de código.		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón que de la parte superior derecha que pone “Welcome [nombre de usuario]”. 3. En el listado de agentes, pulsar sobre el nombre de uno de ellos. 4. En la parte derecha de la ventana, sección “Versions” pulsar sobre el botón “View” de alguna de ellas.		

4.1.3.3 Batalla

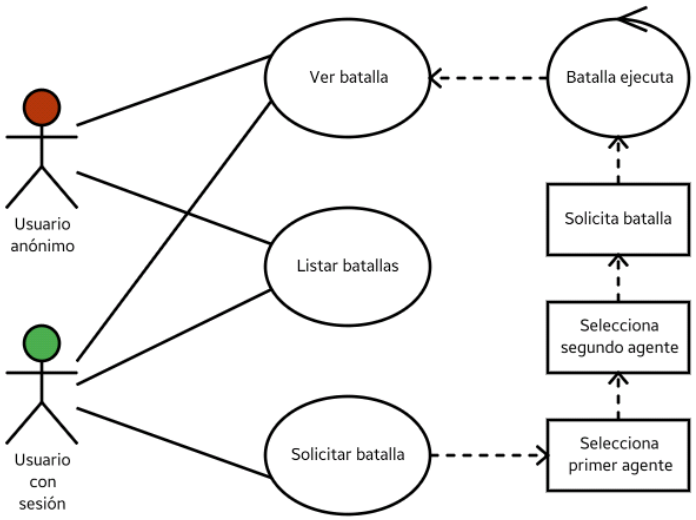


Ilustración 10 *Diagrama de casos de uso en referencia al objeto Batalla*

Caso de uso	Ver batalla	Identificador	R-B-001
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Reproducir una batalla ya ejecutada en el sistema.		
Precondiciones	Al menos una batalla registrada en el sistema		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón “Last Battles” del menú superior. 3. Pulsar sobre los nombres de los agentes participantes en las batallas.		

Caso de uso	Listar batallas	Identificador	R-B-002
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Listar las batallas ya ejecutadas en la plataforma		
Precondiciones	Al menos una batalla registrada en el sistema		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar el botón “Last Battles” del menú superior.		

Caso de uso	Solicitar batalla	Identificador	R-B-003
Actores	Usuario con sesión		
Descripción	Solicitar la ejecución de una batalla entre dos agentes		
Precondiciones	Al menos dos agentes distintos seleccionados para la batalla		
Pasos a seguir	1. Acceder al sitio web. 2. Acceder a una página de usuario y pulsar “Add battle” en el agente deseado. 3. Acceder a una página de usuario y pulsar “Add battle” en el agente deseado por segunda vez. 4. En la parte inferior de la página, pulsar el botón “FIGHT”.		

4.1.3.4 Torneo

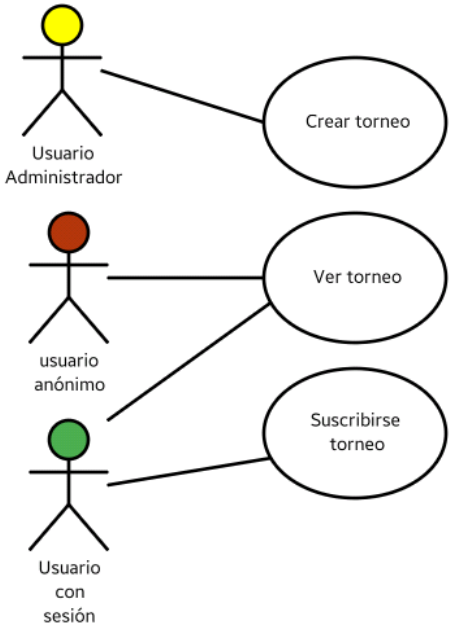


Ilustración 11 *Diagrama de casos de uso en referencia al objeto Torneo*

Caso de uso	Crear torneo	Identificador	R-T-001
Actores	Usuario administrador		
Descripción	Crear un torneo para que los usuarios puedan registrarse		
Precondiciones	Ninguna.		
Pasos a seguir	1. Acceder por SSH al servidor 2. Ejecutar el comando 'node createTournament.js'		

Caso de uso	Suscribirse torneo	Identificador	R-T-002
Actores	Usuario con sesión iniciada		
Descripción	Registrarse en un torneo para participar durante su ejecución.		
Precondiciones	Al menos un torneo creado en la plataforma y abierto a inscripciones.		
Pasos a seguir	1. Acceder al sitio web. 2. Acceder a la sección "Tournaments" del menú superior 3. Hacer click en cualquiera de los botones de "Join Now"		

Caso de uso	Obtener información torneo	Identificador	R-T-003
Actores	Usuario con sesión iniciada, Usuario anónimo		
Descripción	Obtener toda la información de un torneo: Nombre, estado, rondas, resultado de las rondas y batallas asociadas.		
Precondiciones	Ninguna.		
Pasos a seguir	1. Acceder al sitio web. 2. Acceder a la sección “Tournaments” del menú superior 3. Hacer click en cualquiera de los títulos de torneo de la lista		

4.1.3.5 Documentación

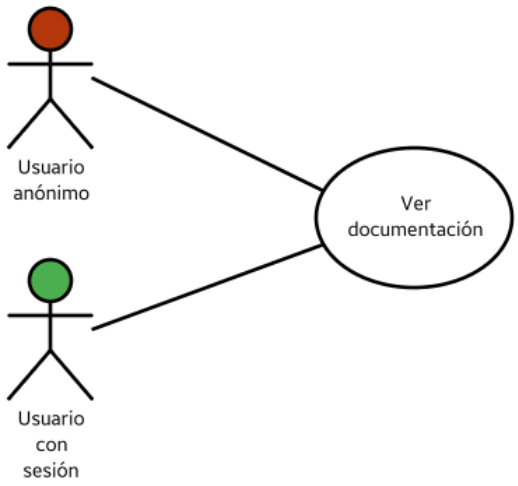


Ilustración 12 *Diagrama de casos de uso en referencia al concepto Documentación*

Caso de uso	Ver documentación	Identificador	R-DO-001
Actores	Usuario con sesión, Usuario anónimo		
Descripción	Acceder a la documentación disponible en el sitio web		
Precondiciones	Ninguna		
Pasos a seguir	1. Acceder al sitio web. 2. Pulsar en el botón “Documentation” del menú.		

4.2 DISEÑO

Como paso previo para un análisis detallado de las arquitecturas concretas del subsistema cliente y el subsistema servidor, en este punto analizaremos las arquitecturas web disponibles y elegiremos una, puesto que esta decisión definirá en gran medida como serán los subsistemas.

4.2.1 Arquitecturas web disponibles

El proyecto se enmarca dentro de lo que podemos denominar un desarrollo web o aplicación web, puesto que el cliente interactúa con el servidor mediante un explorador web en su máquina. Debido a esto, se tendrán en cuenta solo aquellas arquitecturas relacionadas con los desarrollos web.

La esencia de las aplicaciones web, como ya hemos visto, es que las interacciones realizadas por el usuario se realizan siempre a través de un explorador. Aún con esto, hay distintas formas de que estas interacciones acaben alcanzando el servidor, y por tanto, realizando alguna acción en este.

A continuación vamos a realizar un pequeño análisis de las dos arquitecturas principales.

4.2.1.1 Multiple page application

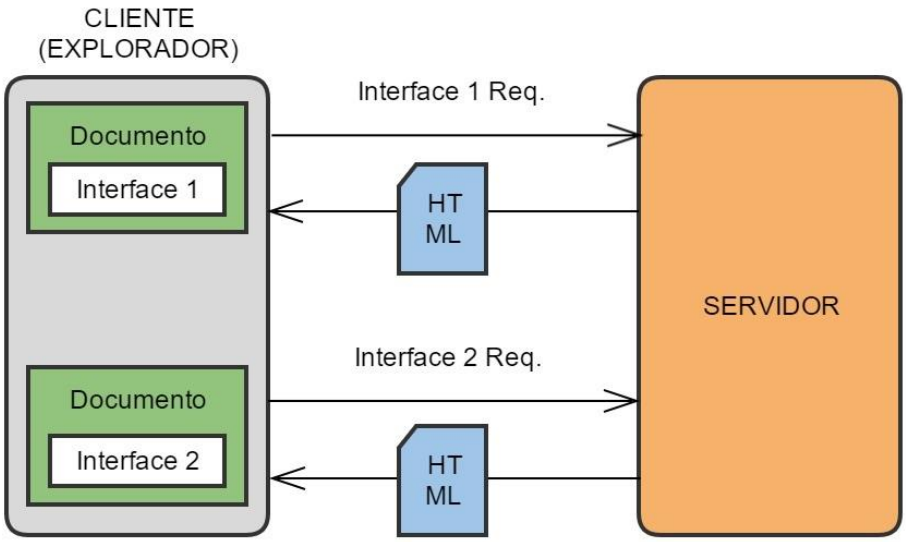


Ilustración 13 *Diagrama de flujo aplicaciones 'Multiple page application'*

Desde la creación de la World Wide Web (WWW), allá por 1990,. En los últimos años, desde 2007 aproximadamente, esta arquitectura se viene abandonando en favor de la arquitectura **Single page application**.

Desde los primeros años de la WWW (El periodo coloquialmente llamado Web 1.0), y hasta hace unos pocos, esta ha sido la arquitectura predominante por resultar extremadamente práctica y

efectiva. Por lo general, los sitios web apenas tenían unas 200 líneas de HTML, no usaban hojas de estilos CSS grandes (En muchos casos no se usaban) y los desarrollos en JavaScript eran poco frecuentes. Como resultado, el peso total de cualquier página solicitada era pequeño. Pero, más importante aún es que, *casi la totalidad de los sitios web, eran estáticos*, y no disponían de ninguna interacción para el usuario. Por esto sobre todo, no era un problema que las acciones de estos desembocaran en una carga completa de un documento.

Con la llegada de la Web 2.0, el tipo mayoritario de sitio web cambió. En lugar de sitios web estáticos, empezaron a proliferar sitios web que almacenaban y ofrecían a los usuarios grandes cantidades de contenido (Algunos de esto son Youtube, Dropbox, Flicker…). Todo esto además con mayor carga en todo lo relativo a la presentación (CSS, HTML5, JavaScript). Fue en esta época en la que se introdujeron además muchas de las tecnologías que harían posible la arquitectura ***Single page application.***

4.2.1.2 Single page application (SPI)

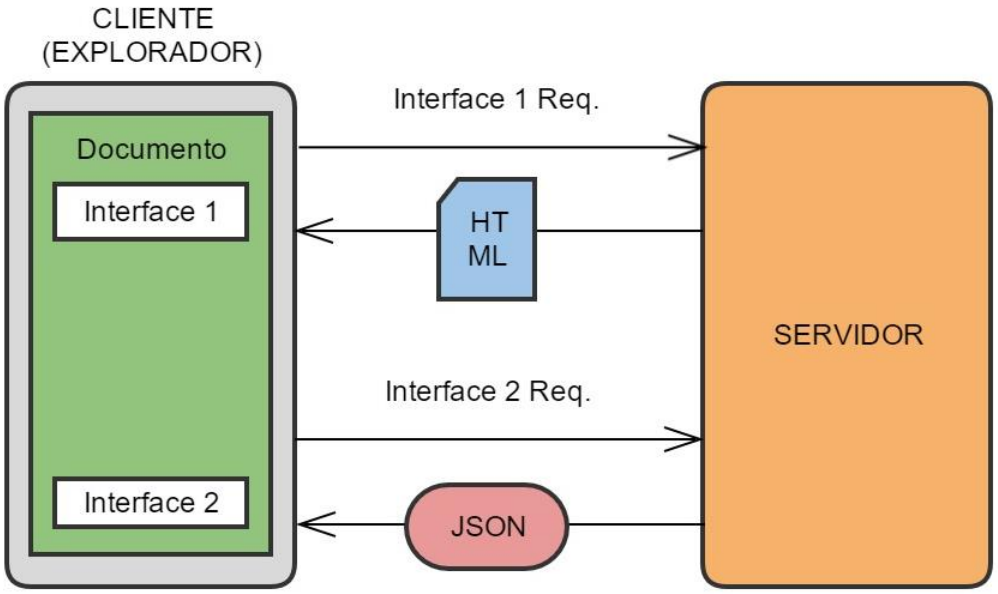


Ilustración 14 *Diagrama de flujo aplicaciones 'Single page application'*

Mientras que en el enfoque clásico, casi todas las acciones del usuario se transformaban en una solicitud de un documento completo nuevo, en este enfoque se carga un único documento que mediante JavaScript y peticiones asíncronas de datos va siendo modificado en función a como el usuario interactúa con el sitio web.

Una de las principales ventajas de esta arquitectura es la mejora de experiencia de usuario, principalmente debido a la reducción en el tiempo de respuesta ante las acciones del usuario. La reducción de este tiempo de respuesta se debe a que:

- No es necesario procesar un documento por acción.
- Sólo se intercambia la información necesaria con el servidor.

La información solicitada por el cliente y que el servidor devuelve, puede venir en dos formatos: JSON y XML. Ambos formatos son estándar. XML ha sido usado durante muchos años para comunicación de sistemas cliente-servidor, pero con la aparición de los sistemas móviles en los que premia la reducción del tráfico de red, JSON superó fácilmente a XML debido a que su formato es mucho más óptimo (para representar exactamente la misma información y con la misma estructura JSON necesita muchos menos Bytes). Para integrar los datos recibidos del servidor de manera asíncrona en el propio documento, se usa JavaScript. Por lo general, una vez que JavaScript ha hecho la petición y esta ha respondido, lee los datos y los inserta en algún nodo del documento HTML. En este proceso pueden intervenir diversos *frameworks*.

4.2.2 **Arquitectura web elegida**

Después de analizar las dos arquitecturas web básicas, se ha tomado la determinación de elegir la arquitectura **"Single page application (SPI)"**. Los motivos de la elección no recaen en necesidades concretas de este proyecto, si no en las ventajas relativas a la experiencia de usuario y a la gestión del proceso de desarrollo que esta arquitectura ofrece para cualquier desarrollo web.

Estas ventajas son:

- Permite reducir al máximo las responsabilidades del servidor en cuanto al aspecto de la aplicación. Estas responsabilidades son delegadas al cliente web. Además, esto tambien facilita una posible integración de otros canales como aplicaciones móviles o Smart TV sin modificar el servidor.
- Los tiempos de carga a los que el usuario es sometido se ven reducidos drásticamente gracias a la reducción de datos enviados y recibidos a través de la conexión a internet.
- Permite gestionar el desarrollo de la parte cliente y la parte servidora como proyectos distintos, cada uno con su control de cambios y empaquetado diferentes.

En contrapartida, deberemos tener en cuenta que la arquitectura cuenta con las siguientes desventajas:

- La gran mayoría del contenido se presenta al usuario mediante ejecución de código JavaScript, por lo que los motores de búsqueda no son capaces de obtener ese contenido. Esto complica mucho la tarea de optimizar el sitio web para motores de búsqueda. Aun así, hay algunas herramientas que permiten ofrecer el contenido ya procesado desde el servidor para solucionar esta problemática. Nosotros usaremos *prerender.io*, para más información, ver el punto Posicionamiento en motores de búsqueda.

- Aunque para algunas cosas manejar el cliente y el servidor como proyectos distintos es positivo, se sigue manteniendo una gran dependencia entre ambos: La comunicación es mediante una API previamente definida. Un cambio en el servidor puede implicar un cambio de API importante, lo cual afecta directamente a la parte cliente (web u otros posibles canales). Es muy importante mantener actualizada la API del servidor.

5 SUBSISTEMA CLIENTE

En este punto vamos a ver todos los detalles de análisis y diseño del subsistema cliente. Este subsistema se ejecuta en el explorador del cliente y tiene la responsabilidad de proveer al usuario de una interfaz que ponga a su disposición todo lo que los requisitos y casos de uso exigen al sistema.

5.1 ELECCIÓN DE TECNOLOGÍAS

En este apartado vamos a exponer todas las tecnologías que han sido elegidas para realizar el desarrollo del cliente. En cada una de ellas, se expone brevemente las razones de su elección para este proyecto.

5.1.1 HTML

HTML (HyperText Markup Language) es un lenguaje de marcado que permite la maquetación de páginas web. Está basado en XML y su primera versión fue publicada en 1991 (Wikipedia, HTML, s.f.) con un total de 18 elementos distintos. La versión actual (5) fue publicada el 24 de octubre de 2014 (Hickson, y otros, s.f.) y cuenta ya con un total de 101 elementos distintos.

Mientras que la primera versión de HTML sólo buscaba organizar la estructura del documento, la versión 5 cuenta con elementos que cumplen tanto una función de estructuración de documento como una función semántica para motores de búsqueda, e incluso lectores de pantalla para personas con discapacidad.

Actualmente HTML es el único estándar para marcado de sitios web, lo cual limita la posibilidad de valorar alternativas distintas. En cualquier caso, HTML cumple con todas las necesidades del proyecto. Como dato curioso, también se puede considerar Flash como sustituto de HTML en su función de maquetación, pero aún en caso de usar Flash, sería necesario usar varios elementos HTML para poder cargarlo.

5.1.2 CSS3

CSS (Cascade Style Sheets) es un lenguaje creado por la W3C (World Wide Web Consortium) que tiene como finalidad definir la manera en la que se presentarán los documentos estructurados mediante HTML. La primera versión de CSS fue publicada por la W3C en diciembre de 1996. La última versión, la 3, fue publicada en noviembre de 2011. (Wikipedia, Hojas de Estilo en Cascada, s.f.).

En las primeras versiones de HTML, los propios elementos dentro del documento contaban con atributos que permitían definir el aspecto que debería tener cada uno de los elementos. En versiones posteriores todos estos atributos han ido desapareciendo en favor de CSS. La idea

principal detrás de la creación de CSS es la separación de responsabilidades entre estructura (HTML) y aspecto (CSS).

CSS es el único lenguaje que existe para definir el aspecto de la estructura HTML de una página web. Actualmente existen algunos metalenguajes como SASS o LESS, que permiten generar CSS con muchas clases en pocas líneas. En este caso se ha decidido no usar ninguno de ellos puesto que estos metalenguajes requieren un tiempo de adaptación que solo los hace viables cuando existe la necesidad de diseñar sitios con gran cantidad de estilos.

Para este proyecto la elección de CSS se debe a que es el único lenguaje de hoja de estilos estándar.

5.1.3 **JavaScript**

JavaScript es un lenguaje de programación interpretado que fue creado en el año 1995 por Netscape Communications Corp. Aunque lo pueda parecer por su nombre, JavaScript no tiene ninguna relación con java. Ambos lenguajes son completamente distintos y parten de premisas muy distintas. Lo cierto es que el hecho de incluir java en el nombre fue una decisión puramente de marketing, puesto que en el momento en el que se presentó, java estaba de moda. (Wikipedia, JavaScript, s.f.)

Cuando hablamos de programación web, es inevitable hablar de JavaScript. Actualmente JavaScript es el único lenguaje de programación web interpretado nativamente por los exploradores. Existen otras alternativas para crear sitios web ricos y dinámicos, como Flash, pero todos requieren de un software adicional que debe ser instalado y que corre empotrado en el explorador, limitando su compatibilidad. Además este tipo de alternativas cuentan con otro problema, y es que todas son propietarias y no cumplen ningún estándar. No así JavaScript, que ya va por su sexto estándar (ECMAScript 6).

En este proyecto, la elección de JavaScript se debe a que es el único lenguaje de programación web en auge y con estándares definidos y apoyados por la comunidad. Toda la potencia de JavaScript no se puede explotar sin HTML5 y CSS3.

5.1.4 **AngularJS**

AngularJS es un framework de JavaScript de código abierto creado por Google (Wikipedia, AngularJS, s.f.) que facilita la creación de aplicaciones web SPI (Single Page Interface). Este framework está basado en el concepto MVC (modelo-vista-controlador). La arquitectura de AngularJS tiene como objetivos:

- Separar de manera eficaz el control del modelo de negocio y el control de los elementos del documento HTML (las vistas).
- Facilitar la creación de test automatizados que garanticen la calidad del código generado.

- Forzar a los desarrolladores a crear desarrollos que sigan buenas prácticas (definidas por Google).
- Proveer todas las herramientas para garantizar que las aplicaciones web pueden tener una separación cliente-servidor efectiva.

Una de las herramientas que permitn a AngularJS ser unos de los framework mejor valorados actualmente, es lo que se conoce como *Two Way Data Binding*. Para entender que es, vamos a compararlo con su conceptualización opuesta, el *One Way Data Binding*.

One Way Data Bindind

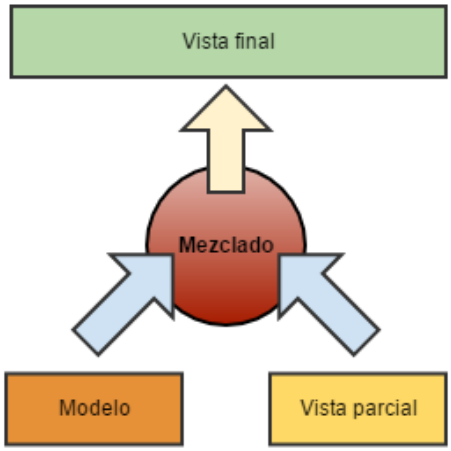


Ilustración 15 *Flujo 'One Way Data Bindind'*

Two Way Data Binding

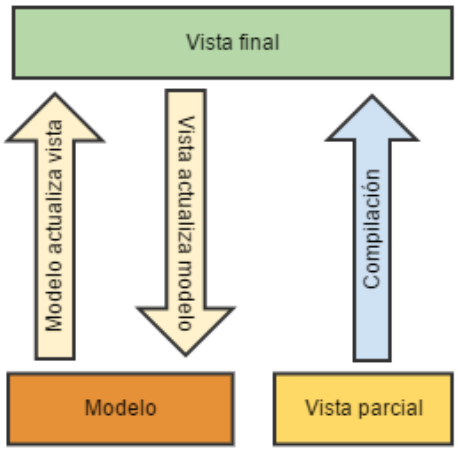


Ilustración 16 *Flujo 'Two Way Data Bindind'*

El modelo de datos y la vista final no tienen ninguna interacción por defecto. Todo cambio en el modelo, debe reflejarse de manera imperativa en la vista y viceversa. Esto resulta tedioso porque obliga a implementar manualmente todos y cada uno de los comportamientos del sistema. Además, para evitar código duplicado, se requiere una gran modularización de cierto código que no aporta nada a la lógica de negocio. Un ejemplo de este enfoque son las aplicaciones que solo usan la librería JQuery.

Las interacciones entre el modelo y las vistas ya vienen resueltas. Si bien resulta necesario declarar como el modelo interactúa con la vista y viceversa, no es necesario programar el código a bajo nivel de estas interacciones (ya que este viene integrado en los framework). La gran ventaja de framworks con este sistema, es que se reduce drásticamente el tiempo a dedicar para desarrollar las interacciones datos-vista, dejando todo ese tiempo disponible para poder diseñar y desarrollar funcionalidad útiles para la lógica de negocio.

5.1.5 KineticJS

KineticJS es una librería de generación de gráficos que se apoya en el elemento *<canvas>* de HTML5. Se usará para la reproducción de partidas. El elemento *<canvas>* de HTML5 (que es el que usaremos para dibujar la interfaz de juego) tiene mucho potencial, tanto en su modo 2D como en el modo 3D, pero controlarlo adecuadamente es un proceso complejo si no se tiene mucha experiencia con él. Para evitar dedicar demasiado tiempo a controlar desde bajo nivel el *<canvas>*, se ha optado por usar KineticJS, que ya cuenta con un conjunto muy grande de funciones sencillas que permiten crear formas y moverlas por el espacio de trabajo. Actualmente la librería ha dejado de tener soporte por parte de su desarrollador, pero la última versión es muy estable y puede ser usada en producción sin ningún problema.

5.1.6 RequireJS

RequireJS es un cargador de ficheros JS especialmente diseñado para ejecutar en desarrollos JavaScript del lado de cliente. Esta tecnología elimina la necesidad de tener que añadir un elemento *<script>* para cada uno de los ficheros JS que necesitamos cargar en el explorador en tiempo de ejecución. Funciona como un cargador modular: mediante código dentro de cada fichero, se especifica las dependencias de este y RequireJS se asegura de que antes de ejecutarlo, todos los ficheros de los que depende han sido a su vez ejecutados.

Otra de las grandes ventajas de RequireJS es que cuenta con un compilador que es capaz de descubrir y trazar todas las dependencias de un proyecto JS y empaquetarlas en un solo fichero minificado (eliminando todo lo innecesario, para ahorrar peso). Para ello utiliza las anotaciones que se hacen previamente para definir las dependencias. Como norma general, durante el desarrollo, RequireJS importa las dependencias descargadas por Bower y las internas del propio proyecto. Una vez que hay una versión lista, se lanza el compilador que genera un único fichero JavaScript con todas las dependencias para subirlo a producción.

En este proyecto usaremos RequireJS por la aportación a nivel semántico (Las dependencias se ven mucho más claras al ver el código) y por la funcionalidad del minificador, que puede llegar a reducir el peso de todo el código JavaScript en producción un 75%.

5.1.7 Grunt

Grunt es un gestor de tareas para desarrollos. Si bien no aporta funcionalidad al producto final, realiza una gran labor ahorrando tiempo a los desarrolladores y diseñadores al permitir automatizar tareas repetitivas. Estas tareas pueden ser de minificación, compilación, ejecución de test, etc… Grunt funciona con un fichero de configuración que se encuentra sincronizado en el repositorio de código del mismo proyecto.

Una de las alternativas más fuertes en su campo es Gulp. Su funcionamiento y utilidades son bastante parecidas.

En este proyecto vamos a usar Grunt para hacer de manera automática la compilación y minificación de todos los ficheros JS del proyecto previamente a su despliegue en producción.

5.1.8 **Bower**

Bower es un gestor de dependencias para desarrollos webdel lado del cliente en Javascript. La ventaja de este sistema es que simplifica la instalación de una dependencia de JavaScript a una simple línea de consola. Funciona a través de un fichero de configuración dentro del proyecto y que se sincroniza en el repositorio de código. Cada vez que un desarrollador se descarga el proyecto en su equipo, su instalación de bower verifica el fichero de configuración del proyecto e instala todas las dependencias desde un repositorio global. Las ventajas principales son:

- Simplifica la instalación y actualización de dependencias teniendo en cuenta la versión de estas.
- Evita tener que subir todas las dependencias al repositorio de GIT, lo que suele ser un problema debido a su tamaño y al control de cambios.
- No altera el proceso normal de importación de dependencias desde HTML o RequireJS

En este proyecto usaremos Bower para simplificar la instalación y gestión de dependencias cuando se trabaja desde distintos equipos.

5.2 ARQUITECTURA

En este punto vamos a analizar la arquitectura del subsistema cliente, viendo como los componentes de este se relacionan entre si y la función de cada uno.

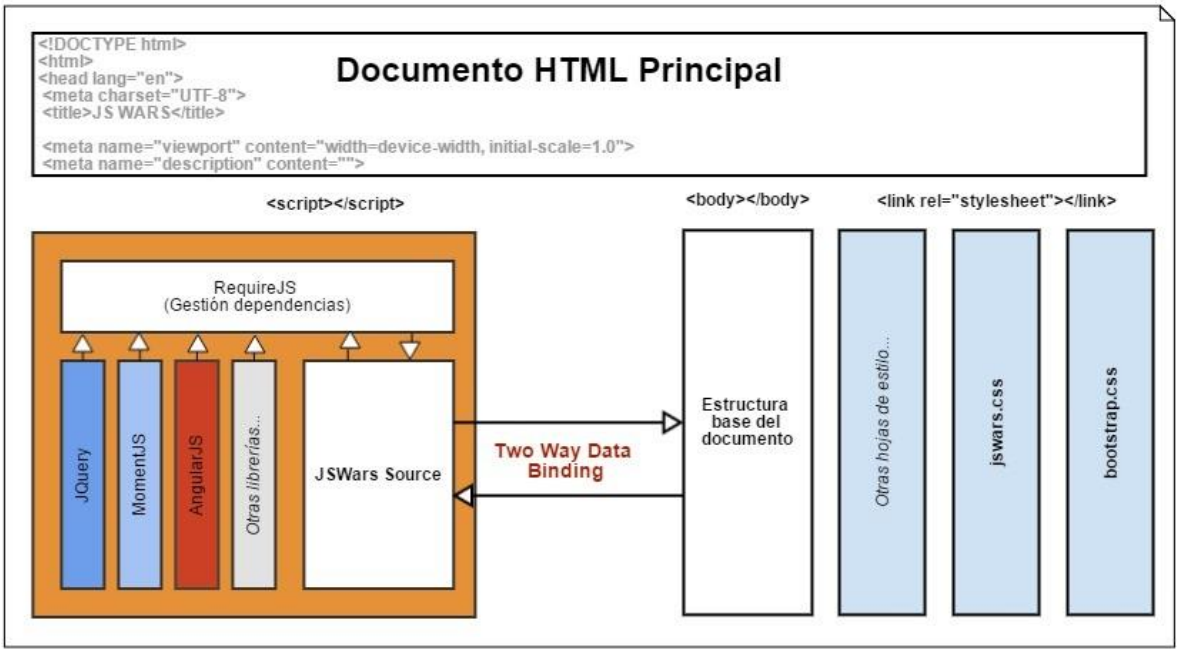


Ilustración 17 *Arquitectura del cliente*

La arquitectura del subsistema cliente está distribuida en una gran cantidad de ficheros JavaScript que son importados cuando son necesarios mediante RequireJS, encargado de gestionar las dependencias. Pero no es RequireJS el que define los tipos de componentes JavaScript que tenemos, si no AngularJS, que es sobre lo que realmente el sistema se apoya y sobre lo que se encuentra el 100% del código JavaScript desarrollado dentro del cliente (RequireJS no aporta funcionalidad, solo herramientas para gestionar dependencias).

AngularJS tiene un diseño conceptual que define los siguientes tipos de componentes, todos ellos pueden ser creados por el programador (En este proyecto cada uno de los componentes es una dependencia de RequireJS distinta):

- **Templates**: Fichero HTML que se usará para dar el aspecto a una vista. Puede incluir *angular expressions* que permitirán después agregar datos.
- Directivas: Tienen muchas funciones distintas que van desde alterar el comportamiento natural de los elementos ya existentes en HTML hasta la creación de nuevos elementos específicos, pero siempre operan con elementos del documento. En todos los casos las directivas encapsulan funcionalidades adicionales al original del elemento (el alcance de estas es cualquier comportamiento que se pueda programar por JavaScript).
- **Filtros**: Se usan dentro de las llamadas *angular expression*. Su función principal es decorar cualquier dato que se desee pintar en la vista desde el controlador. Un ejemplo clásico de

uso de filtros es la moneda (En el modelo tenemos un '5', si lo imprimimos aplicando el filtro *currency* se agrega a la vista como '5 €'). Ayuda mantener la responsabilidad del controlador centrada en funcionalidad evitando a su vez código duplicado en los templates.

- **Controladores**: Se ocupan de proveer datos a la vista (template ya procesado por el compilador) y manejarla. La comunicación con la vista es mediante el *two way data binding* (Ver AngularJS para más información). Este además es el que controla el *scope*, que no es más que el contexto sobre el que opera el *two way data binding* en su proceso continuo de observación del modelo.
- **Servicios**: Tienen como responsabilidad controlar la lógica de negocio, especialmente si existe mucha que es común a muchos controladores. Normalmente, si la aplicación obtiene datos de un servidor, como es este caso, se apoya en los *Factories*.
- **Factories**: Los *Factories* (componentes factoría) tienen como responsabilidad la comunicación directa con servidores remotos. Por buenas prácticas, Su funcionalidad solo debe ser usada por los servicios. La diferencia principal entre los *servicios* y los *factories* es que los primeros controlan la lógica de negocio, y los segundos solo tienen conocimientos de los *objetos del negocio* y de cómo realizar las operaciones de creación, modificación, obtención y borrado contra el servidor.
- **Módulos**: Su función es únicamente constituir paquetes del resto de componentes: *directivas*, *filtros*, *controladores*, *servicios* y *factories*. No aportan funcionalidad como tal al sistema.

AngularJS ya cuenta con algunos componentes empotrados, pero el desarrollador final siempre tiene que desarrollar más. Todos estos componentes se pueden obtener de la comunidad y se importan mediante RequireJS.

5.2.1 Componentes desarrollados

En este punto vamos a analizar los componentes relevantes desarrollados específicamente para el proyecto, todos ellos integrados dentro del contexto de AngularJS.

5.2.1.1 Directivas

Las directivas tienen como principal función encapsular dentro de un solo elemento HTML una gran cantidad de lógica, e incluso aspecto. La gran ventaja de este tipo de componentes es que son reutilizables en cualquier *vista* y no requieren ninguna modificación de funcionalidad en el *controlador* de esta.

5.2.1.1.1 GameDirective

Esta directiva se ocupa de gestionar todo lo relacionado con la reproducción del juego. Para insertar un reproductor de partida en *cualquier* punto de una vista, solo es necesario introducir en un *template* el siguiente código:

```
<div game battle="battle"></div>
```

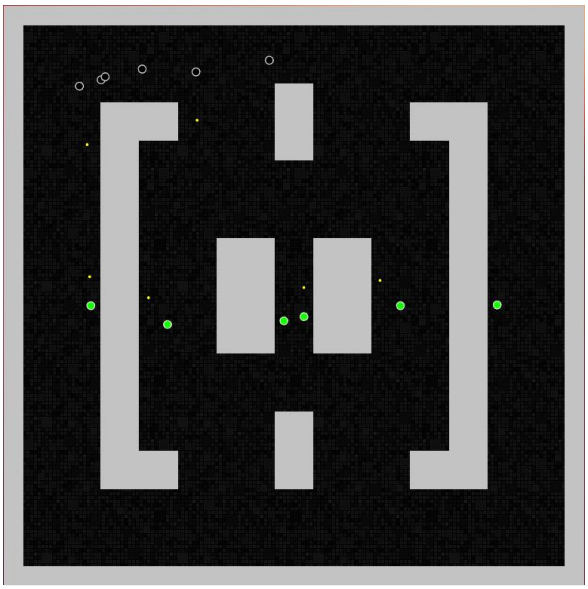


Ilustración 18 *Ejemplo de reproducción de partida mediante GameDirective*

El parámetro *battle* debe de ser el objeto *battle* a reproducir. Este objeto debe ser precargado por el *controlador* de la *vista* (desde el servidor). Además en la definición de la directiva existen una serie de parámetros adicionales que permiten exponen una serie de métodos para controlar la reproducción:

- restart(): Pone el reproductor en fotograma 0 de la partida.
- start(): Activa la reproducción de la partida.
- continue(): Pone a reproducir una partida en pausa.
- next(fotogramas): Permite avanzar un número de fotogramas. La reproducción avanzará o retrocederá según el signo del número de fotogramas.
- pause: Permite pausar la reproducción de la partida.

Y dos eventos:

- on-start: Función a llamar cuando la partida comienza a reproducirse.
- on-frame: Función a llamar cuando se dibuja un fotogramas.

Haciendo uso de todos los métodos y eventos disponibles, se puede insertar un reproductor de partida en cualquier lugar, con o sin controles, o solo con algunos de ellos sin más trabajo que el que supone colocar los botones de control en un HTML.

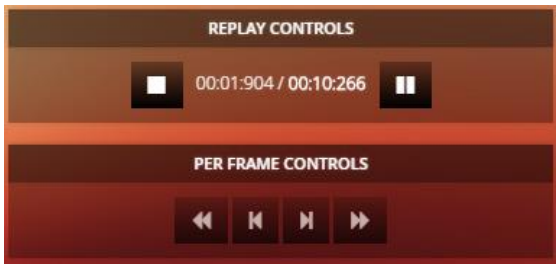


Ilustración 19 *Ejemplo de controles externos en GameDirective*

5.2.1.1.2 BattleCreatorDirective

Esta directiva se ocupa de gestionar el creador de batallas que 'flota' en la parte inferior de cualquier interfaz del subsistema cliente. Solo está visible para usuarios registrados, que son los que pueden crear batallas. El elemento que la encapsula es:

```
<battle-creator></battle-creator>
```

Ilustración 20 *Aspecto de la directiva BattleCreatorDirective con una configuración lista para enviar*

Únicamente cuenta con dos interacciones directas:

- Fight: Crea la batalla. Requiere tener dos agentes configurados para la batalla.
- Clear: Elimina la configuración de batalla actual, permitiendo añadir una nueva.

Esta directiva mantiene de manera persistente en el explorador del usuario la última configuración realizada, y depende de la directiva BattleCreatorButtonDirective para recibir las solicitudes de configuración de los agentes que se desean añadir. La comunicación de ambas directivas se realiza mediante eventos dentro de AngularJS.

Cuando la batalla se ha solicitado con éxito, la directiva muestra una imagen de carga y espera a que la batalla termine de ejecutarse mediante la técnica del *long-pooling.*

Ilustración 21 *Aspecto de la directiva BattleCreatorDirective durante la ejecucón de una batalla*

El *long-pooling* es una técnica usada para obtener eventos en tiempo real a través de conexiones HTTP. Por la naturaleza de las conexiones HTTP, en principio no es posible recibir información sin

solicitarlo previamente, por lo que lo más cercano a una comunicación en tiempo real se consigue haciendo muchas peticiones en un corto espacio de tiempo. Como alternativa a esto nace el concepto de *long-pooling. S*e basa en abrir una conexión HTTP y no contestarla hasta que no se tiene un evento que proporcionar.

En este proyecto en concreto, la conexión HTTP que informa del estado de ejecución de la batalla no se finaliza hasta que la batalla no ha terminado de ejecutar. En el momento exacto en el que termine la ejecución de la batalla, el cliente detectará tal hecho gracias al cierre de la conexión, y su contenido.

5.2.1.1.3 BattleCreatorButtonDirective

Esta directiva se ocupa de convertir un elemento sin funcionalidad en un botón para configurar un agente determinado en una batalla usando el atributo 'battle-creator-button'. Complementa en funcionalidad a la directiva BattleCreatorDirective, puesto que es la que se ocupa de lanzar los eventos de configuración de agente que esta última necesita. Para añadir el comportamiento a un botón, podríamos hacerlo de la siguiente manera.

```
<button type="button" battle-creator-button="{id: agent._id, username:
                user.username}">Add to Battle</button>
```

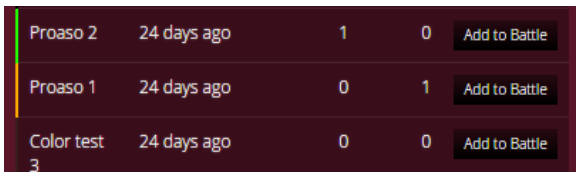


Ilustración 22 *Ejemplo de instancias de la directiva BattleCreatorButtonDirective*

Como único parámetro de inicialización recibe un objeto que contiene el Identificador del *Agente* y el nombre de *Usuario* al que está asociado el agente. Una vez que el usuario pulse el botón, se añadirá a la configuración de la batalla el agente seleccionado.

5.2.1.1.4 ProfileUpdateDialogDirective

Esta directiva permite al usuario realizar una modificación de sus datos relevantes del perfil en formato de ventana modal. La directiva modifica el comportamiento de click de todo elemento cuando este contiene el atributo 'profile-update'. El evento de click del elemento lanza a partir de ese momento una modal con campos para modificar el perfil del usuario actual. Un ejemplo sería:

```
<span profile-update on-complete="profileUpdated”>Modificar</span>
```

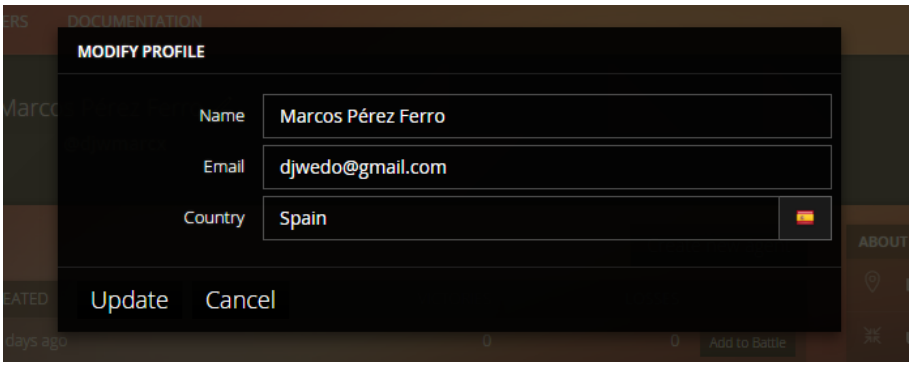


Ilustración 23 *Ejemplo de instancia de la directiva ProfileUpdateDialogDrtv*

La directiva cuenta con un evento:

- on-complete(): Función llamada cuando se cierra la ventana modal. Puede ser útil por ejemplo para solicitar de nuevo los datos de usuario y obtenerlos actualizados.

5.2.1.1.5 TournamentJoinDialogDrtv

Esta directiva permite al usuario registrarse para un torneo concreto. La directiva modifica el comportamiento de click de todo elemento que contenga el atributo 'tournament-join'. El evento de click muestra una modal que permite elegir con que agente se quiere realizar la inscripción al torneo.

```
<button tournament-join="56c8b5ab2666c8741d23aad3”>Join Now</button>
```

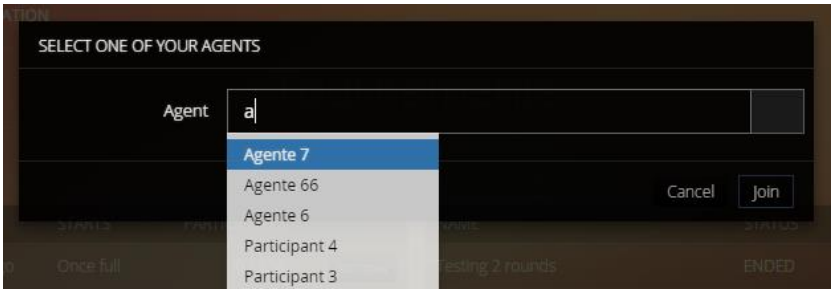


Ilustración 24 *Ejemplo de instancia de la directiva TournamentJoinDrtv*

5.2.1.2 Factories

Los *Factories* tienen la definición de todas las operaciones disponibles en la *API Rest* del servidor. Se agrupan en ficheros teniendo en cuenta el objeto al que se refieren (Agente, Batalla, etc…). Para más información sobre la definición de los servicios, revisar el punto *API Rest*.

El listado concreto de *factories* desarrollados es:

- **SessionFactory:** En este *factory* se encuentran definidas todas las llamadas necesarias para permitir al usuario controlar su sesión actual dentro del sistema. Actualmente solo se hace uso de una de una de ellas:
    - Obtener la sesión actual del usuario

- **UserFactory:** Define el formato de llamadas de las operaciones relacionadas con el usuario. La gran mayoría de estas operaciones alteran su comportamiento o incluso son bloqueadas en función a quien pertenezca la sesión actual. Las operaciones definidas son:
    - Listar usuarios paginados
    - Obtener un usuario
    - Actualizar un usuarios (Solo se puede si la sesión actual se corresponde)

- **AgentFactory:** Define el formato de las llamadas relaciondas a operaciones sobre un agente. No se debe confundir esta parte con la equivalente para gestionar *versiones de agente* (Siguiente punto). Las operaciones definidas son:
    - Listar agentes de un usuario paginados
    - Crear un agente en el usuario actual
    - Modificar un agente del usuario actual

- **AgentVersionFactory:** Contiene la definición de las llamadas relacionados con la gestión de las diferentes versiones de código de un agente. En este caso solo hay definida una operación:
    - Crear versión del agente

- **BattleFactory:** Define el formato de las llamadas necesario para realizar las operaciones relacionadas con batallas. Las operaciones concretas son las siguientes:
    - Obtener información de una batalla
    - Obtener una partición de la partida (*chunk*)
    - Solicitar una batalla
    - Obtener el estado de una solicitud de batalla.

- **TournamentsFactory:** Define como gestionar las llamadas necesarias para realizar las operaciones relacionadas con torneos dentro del sistema.
    - Listar torneos paginados
    - Obtener información de un torneo
    - Suscribirse a torneo

5.2.1.3 Servicios

Los *servicios* contienen normalmente todo aquello relacionado con el modelo de negocio o con funcionalidades compartidas. En este caso la gran mayoría del modelo de negocio convive en el

subsistema servidor, por lo que actualmente los *servicios* únicamente tienen como finalidad facilitar el acceso a los métodos disponibles en las *factorías*. Entre ellos podemos encontrar:

- **AgentService**: Expone *AgentFactory*
- **AgentVersionService**: Expone *AgentVersionFactory*
- **BattleService**: Expone *BattleFactory*
- **SessionService**: Expone *SessionFactory*
- **TournamentService**: Expone TournamentFactory
- **UserService**: Expone UserFactory
- **AnimationFrameService**: Permite una gestión global de las solicitudes de dibujo de fotogramas. Este servicio es usado de manera intensiva por *GameDirective.*
- **StateService**: Gestiona la estructura estándar de títulos en todas las interfaces.

5.2.1.4 Controllers

Los *controladores* son los componentes encargados de gestionar las vistas y su lógica implícita. Tienen como responsabilidad solicitar los datos necesarios, refrescarlos y controlar las paginaciones de estos, además de realizar un pequeño preprocesado de los datos recibidos de los servicios si fuera necesario. Existe uno por cada una de las interfaces implementadas, y la funcionalidad encapsulada dentro de ellos es la que se puede apreciar a nivel usuario, por lo que para más detalle, ver el punto Interfaces implementadas.

5.2.2 Interfaces implementadas

En este punto se explicarán cada una de las interfaces implementadas para dar cobertura a los casos de uso y los requisitos. Las imágenes que se muestran pertenecen a la visualización en *escritorio,* pero todas se ven correctamente en dispositivos móviles, como *tabletas* o *teléfonos.*

5.2.2.1 Listado de batallas

La interfaz *listado de batallas* está compuesta únicamente por un listado paginado de batallas ordenadas en orden descendente con respecto a la fecha de ejecución.

Su función es doble: por un lado, permite al usuario observar que está pasando en la plataforma (actividad reciente) y por otro lado ejerce una función de descubrimiento de contenido. Puesto que en el diseño conceptual del sistema las batallas tienen dependencia de los *agentes*, y estos a su vez

de los *usuarios*, la navegabilidad lógica sería entrar a las *batallas* desde la vista de *usuario* de uno de los usuarios participantes, pero esto resulta confuso y complica la visualización de múltiples batallas.

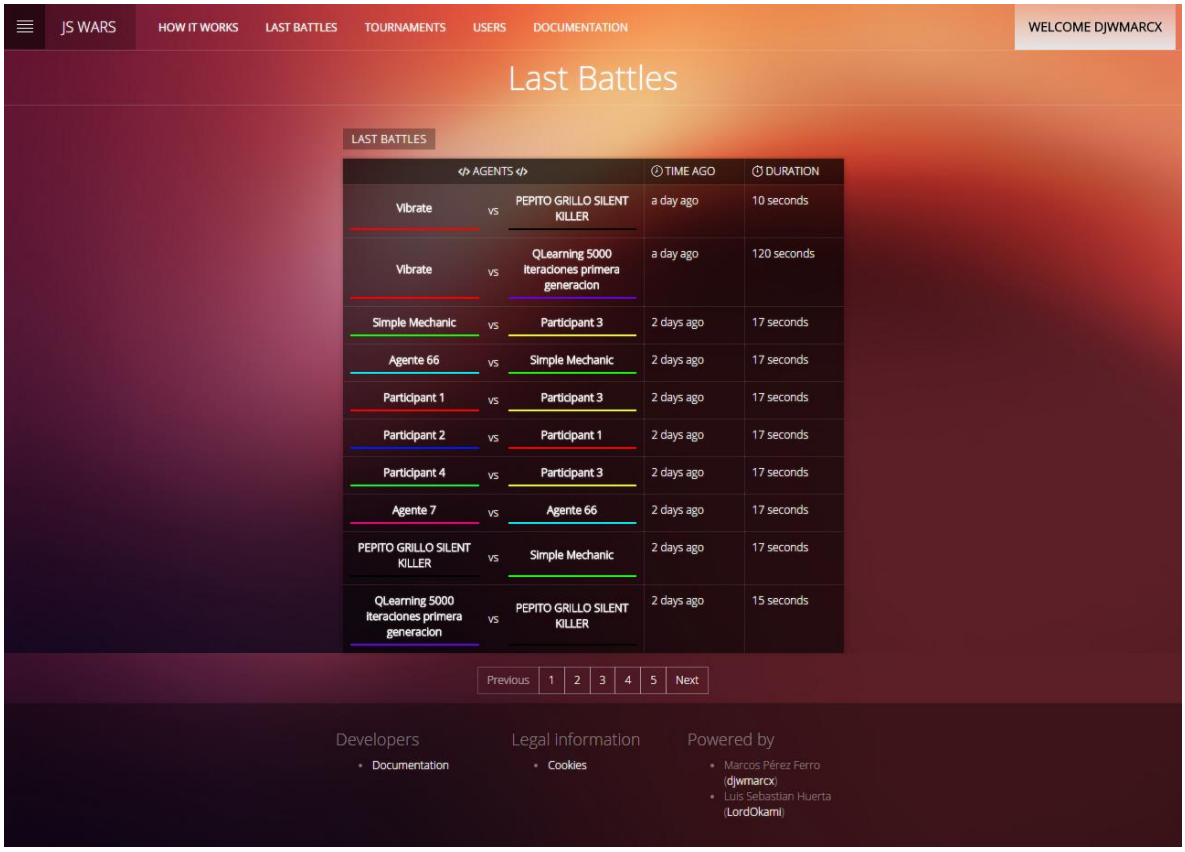


Ilustración 25 *Aspecto de la interfaz 'Listado de batallas'*

El listado de batallas contiene el nombre de los dos agentes participantes, el tiempo que hace que se ejecutó y su duración en segundos. Además los datos de la tabla contienen información adicional que se muestra al pasar por encima de los elementos importantes con el ratón o con un lápiz táctil en una Tablet:

Ilustración 26 Detalle de autor del agente en 'Listado de batallas'

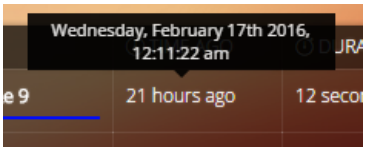


Ilustración 27 Detalle de hora de batalla en 'Listado de batallas'

5.2.2.2 Visualización de batalla

La interfaz *visualización de batalla* es la que permite reproducir una batalla una vez que ha sido ejecutada por el servidor. Es la interfaz más compleja del sistema y se apoya sobre la directiva GameDirective para obtener casi toda su interactividad.

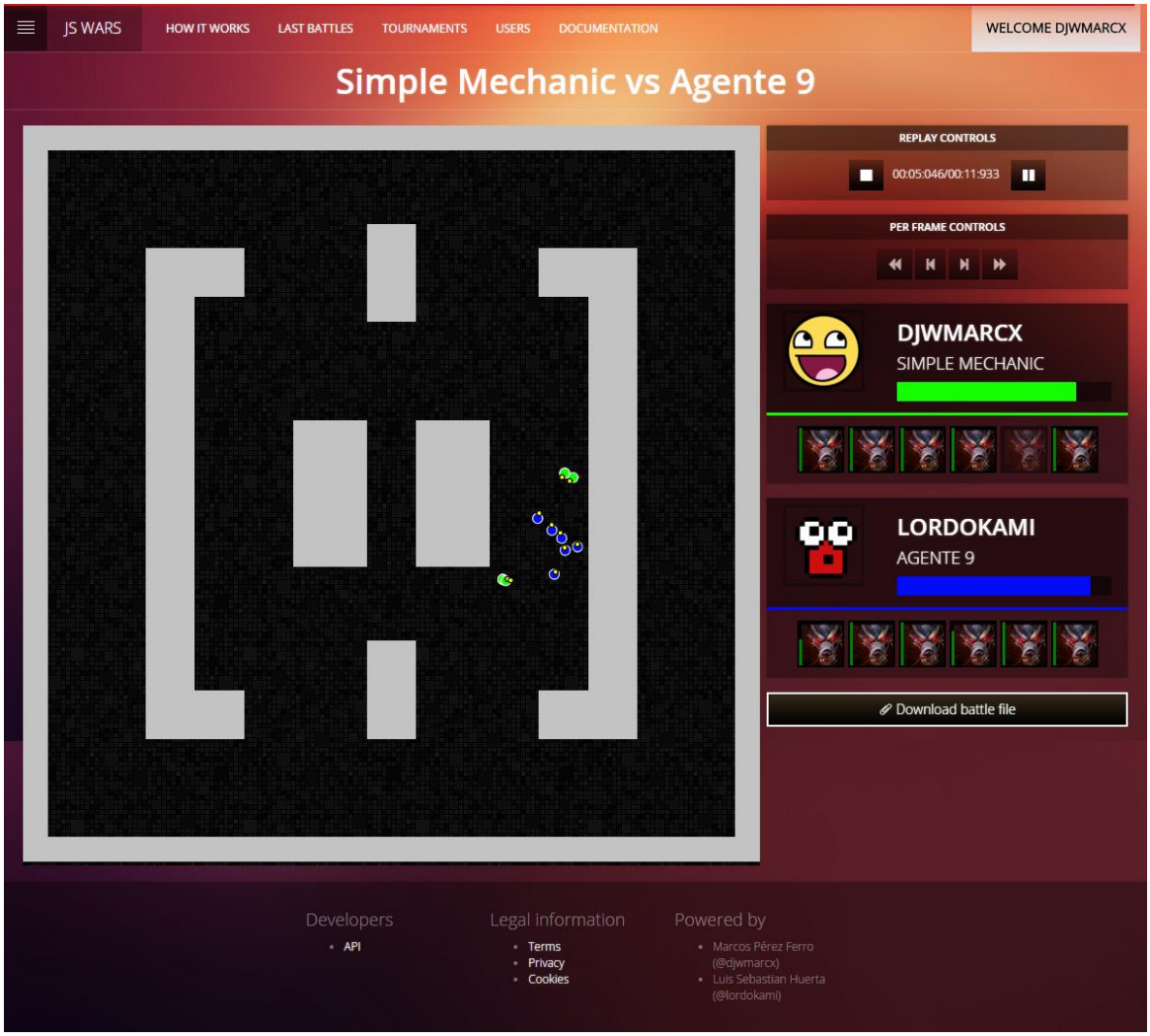


Ilustración 28 *Aspecto de la interfaz 'Visualización de batalla'*

A nivel visual la interfaz se compone de las siguientes zonas:

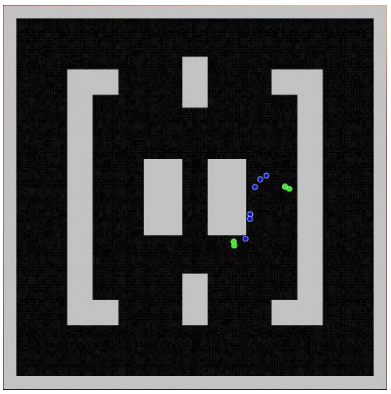


Ilustración 29 Detalle del reproductor en la interfaz 'Visualización de batalla'

Reproductor de batalla: Controlada por la directiva *GameDirective*. Esta parte de la interfaz es la que muestras las el estado de la partida en cada fotograma. El mapa se obtiene del servidor antes de comenzar la partida, y en cada iteración las unidades dibujadas en él se mueven o disparan. El color de las unidades de cada equipo es el que el usuario seleccionó al crear el agente por primera vez.

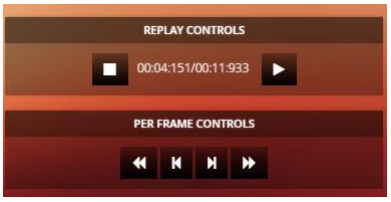


Ilustración 30 Detalle de los controles en la interfaz 'Visualización de batalla'

Controles de batalla: Esta parte de la interfaz tiene como único cometido el mostrar el estado actual de la partida y permitir el control de la ejecución de esta. Entre las herramientas de control que se dan, podemos encontrar las siguientes: Parar, iniciar, retroceder 10 o 1 fotogramas y avanzar 1 o 10 fotogramas.

Ilustración 31 Detalle del estado de equipo en la interfaz 'Visualización de batalla'

Estado de batalla: Esta parte de la interfaz muestra el estado de cada usuario participante en la batalla (Actualmente siempre son 2). Los datos que ofrece esta parte de la interfaz por equipos son: Nombre de usuario, nombre del agente, salud total actual (En formato barra, siendo esto medido como suma total de las vidas de las unidades) y una caja en la parte inferior con la representación de cada unidad y su correspondiente barra de salud (en vertical). En este apartado de la interfaz es donde se aprecia realmente el estado de la partida de un equipo con respecto a otro.

5.2.2.3 Creación de batalla

La interfaz de creación de batalla flota en la parte inferior de todas las páginas mostrando la información de los agentes agregados a la configuración de batalla. Esta interfaz está compuesta únicamente por la directiva BattleCreatorDirective.

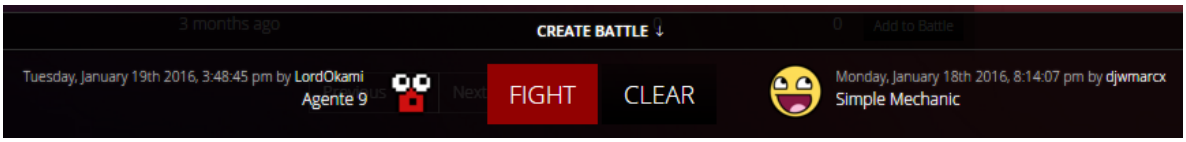


Ilustración 32 *Aspecto de la interfaz 'Creación de la batalla'*

La interfaz tiene los siguientes estados:

- **Oculta:** La interfaz es invisible si ningún agente ha sido agregado a la configuración de la batalla o el usuario es anónimo. En el momento que se agregue un agente, la interfaz pasa al estado *parcialmente completa.*
- **Parcialmente completa:** La interfaz es visible pero no se puede iniciar una batalla (*fight*) porque solo se ha agregado un agente a la batalla. El agente pendiente de agregar se muestra como se ve a continuación.

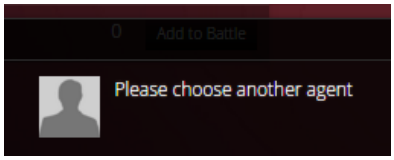


Ilustración 33 *Detalle creación de batalla incompleta en la interfaz 'Creación de la batalla'*

- **Completa:** La interfaz es visible y la batalla se puede iniciar con el botón *fight*.

Los usuarios sin sesión no pueden agregar agentes ni ver esta interfaz.

5.2.2.4 Listado de torneos

La interfaz *listado de torneos* permite obtener listados paginados tanto de los torneos actualmente abiertos como los ya pasados.

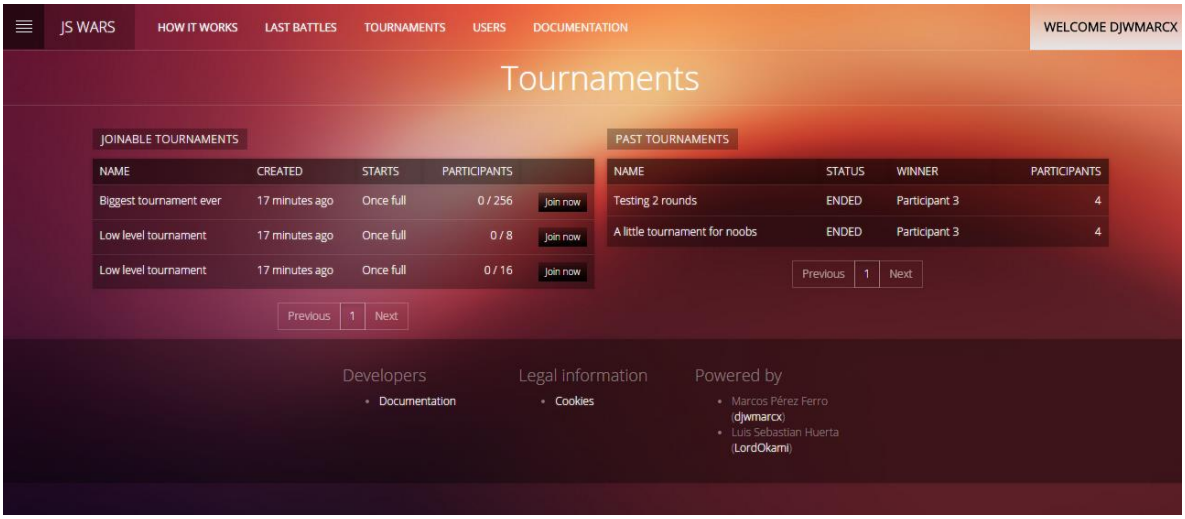


Ilustración 34 *Aspecto de la interfaz 'Listado de torneos'*

La interaz se divide en dos partes:

- **Joinable tournaments**: Son aquellos torneos a los que los usuarios todavía pueden unirse. El hecho de que el torneo esté abierto a entrada de usuarios, no garantiza que un usuario concreto pueda entrar, puesto que si ya se registró, no se le dejará volver a registrarse. (Botón *Join Now* apagado). Los datos que podemos encontrar en este listado son:
    - **Name**: Nombre del torneo.
    - **Created**: Momento de creación del agente.
    - **Starts**: Evento con el que comenzará el torneo (Por ahora siempre es cuando se llena).
    - **Participants**: Cantidad de participantes registrado y cantidad máxima admisible.

- **Past Tournaments**: En esta parte podemos encontrar todos aquellos torneos que ya no se encuentran pendientes de ejecución. Los datos que encontramos son:
    - **Name**: Nombre del torneo.
    - **Status**: Estado del torneo, pueden ser:
        - *ENDED*: Estado ideal, el torneo se ejecutó con éxito.
        - *ERROR*: Hubo un error durante la ejecución del torneo, no se ha terminado.
        - *RUNNING*: El torneo aún está ejecutándose.
    - **Winner:** Agente ganador del torneo.
    - **Participants:** Participantes totales.

En el listado de torneos disponibles para el usuario (Joinable tournaments), se muestra en un globo un detalle de la hora y un botón para apuntarse si todavía es posible (El usuario no está apuntado ya). Al pulsarlo aparece la interfaz Unirse a Torneo.

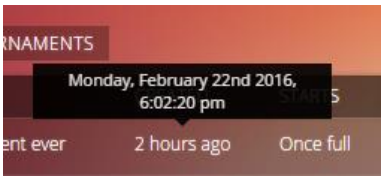


Ilustración 36 *Detalle hora de creación en interfaz 'Listado de torneos'*

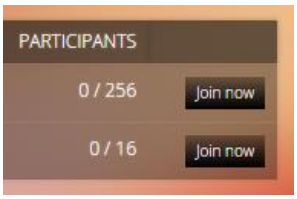


Ilustración 35 *Detalle del botón para unirse al torneo en interfaz 'Listado de torneos'*

5.2.2.5 Unirse a torneo

Esta interfaz se apoya en la directiva *TournamentJoinDialogDrtv* para funcionar. Se compone de una vista flotante que aparece cuando se pulsa en un botón *Join Now* y solicita al usuario un agente con el que unirse al torneo.

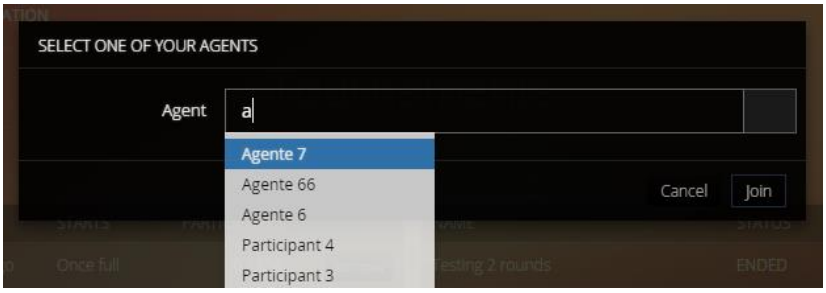


Ilustración 37 *Aspecto de la interfaz 'Listado de torneos'*

El campo que aparece es en realidad un campo de búsqueda, por lo que al escribir una letra o varias, busca entre los agentes del usuario una coincidencia con los nombres.

5.2.2.6 *Detalle de torneo*

Esta interfaz muesta toda la información disponible acerca de un torneo. Puede mostrar información de torneos en cualquier estado, cambiando su disposición automticamente.

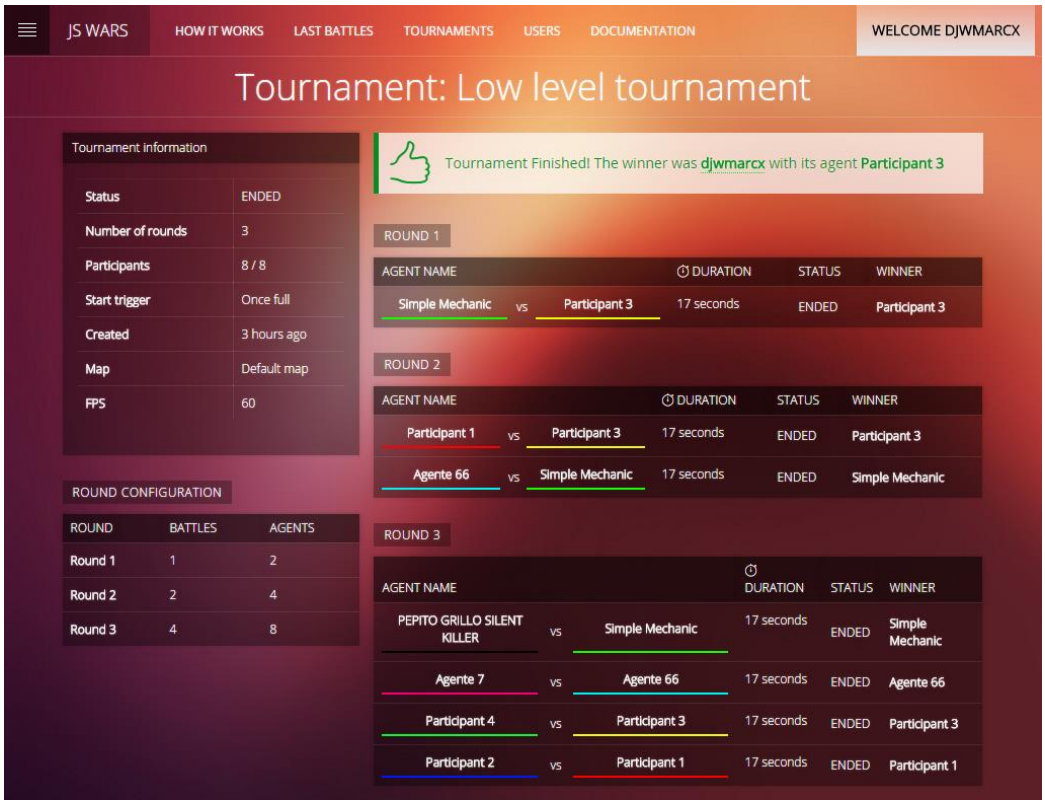


Ilustración 38 *Aspecto de la interfaz 'Detalle de torneo'*

Cuenta con las siguientes partes:

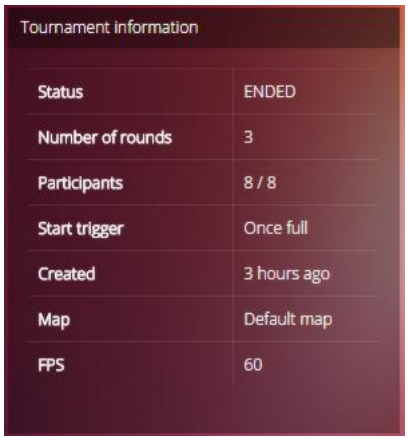


Ilustración 39 *Detalle de información de torneo en la interfaz 'Detalle de torneo'*

Detalles del torneo: Muestra la información básica del objeto torneo (No depende de su estado):

- Estado
- Rondas
- Participantes
- Disparador de inicio
- Fecha de creación
- Mapa
- Fotogramas por segundo

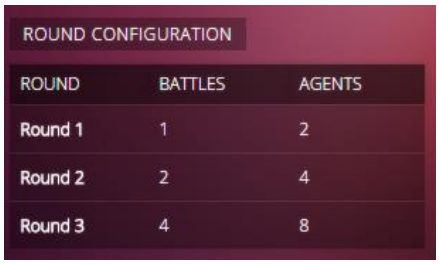


Ilustración 40 *Detalle de configuración de rondas en la interfaz 'Detalle de torneo'*

Configuración de rondas: Contiene un resumen de todas las rondas que se van a realizar y su configuración en cuanto a agentes y batallas de cada ronda. Es importante tener en cuenta que la numeración de las rondas es inversa al orden en el que se ejecutan, siendo la 3 en el caso de la imagen la que primero se ejecuta.

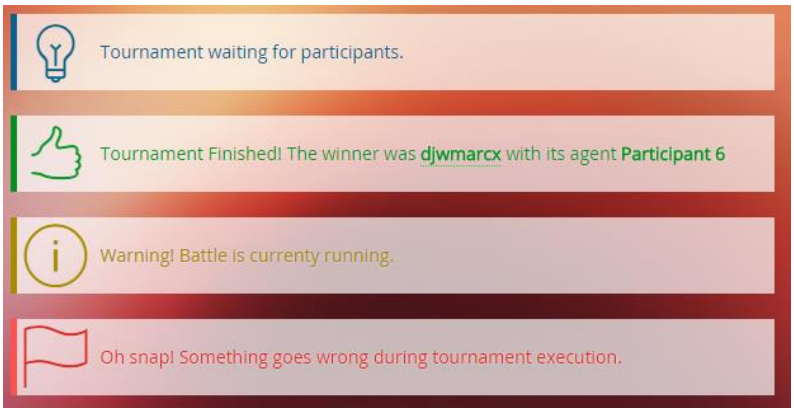


Ilustración 41 *Detalle de indicador de estado de torneo en la interfaz 'Detalle de torneo'*

Indicador de estado: Indica el estado en el que se encuentra el torneo del que se está obteniendo la información. Forma un papel muy importante para permitir entender al usuario porque ve la pantalla como la ve.

Ilustración 42 *Detalle de rondas en la interfaz 'Detalle de torneo'*

Detalle de rondas: Cada una de las rondas ejecutadas en el torneo tiene su propio bloque con el listado de batallas ejecutadas dentro de la ronda, su duración y el ganador. Estos bloques solo aparecen si la batalla está en estado *ENDED*

5.2.2.7 Listado de usuarios

El listado de usuarios permite ver todos los usuarios registrados en la plataforma de forma paginada. Junto a cada usuario, se muestra su fecha de registro, su avatar y la cantidad de agentes almacenados en su nombre.

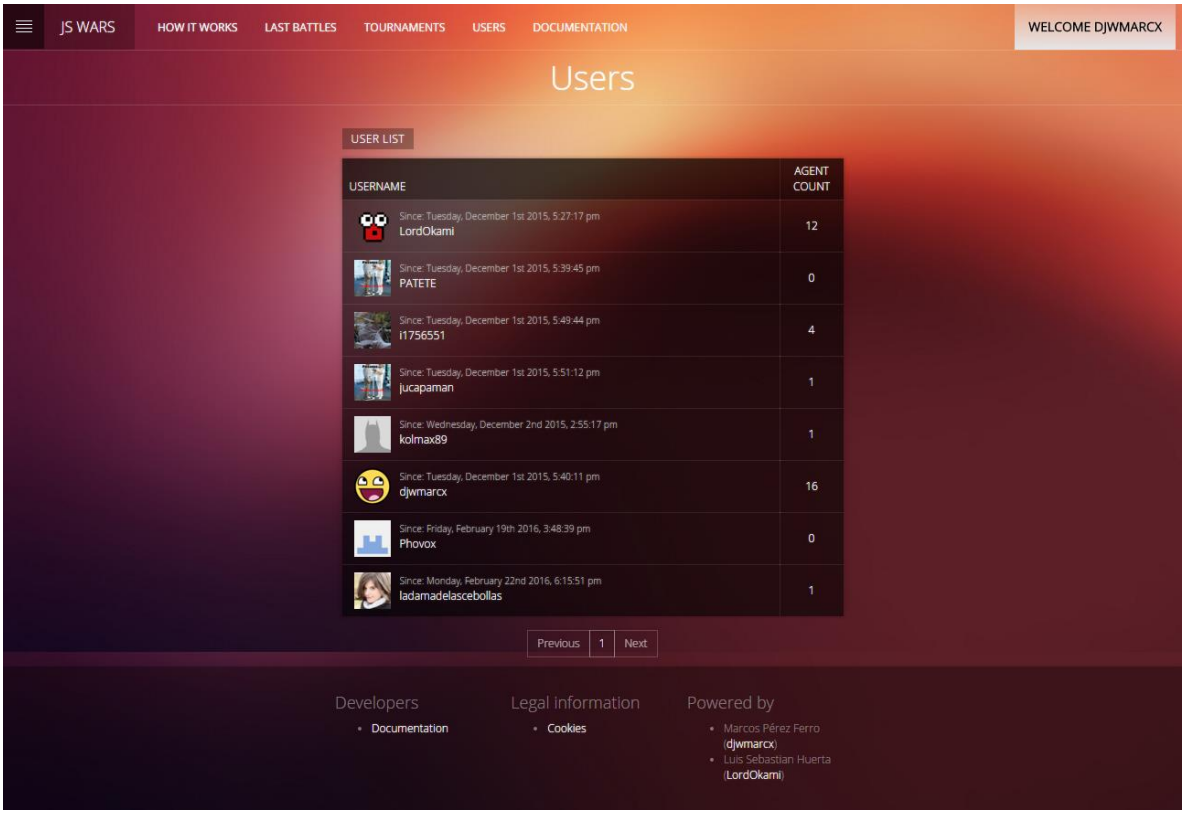


Ilustración 43 *Aspecto de la interfaz 'Listado de usuarios'*

5.2.2.8 Detalle de usuario

La interfaz de detalle de usuario proporciona toda la información de un usuario. Entre la información de interese se encuentran sus datos personales (usuario, nombre, nacionalidad y fecha de registro) además de sus agentes y torneos.

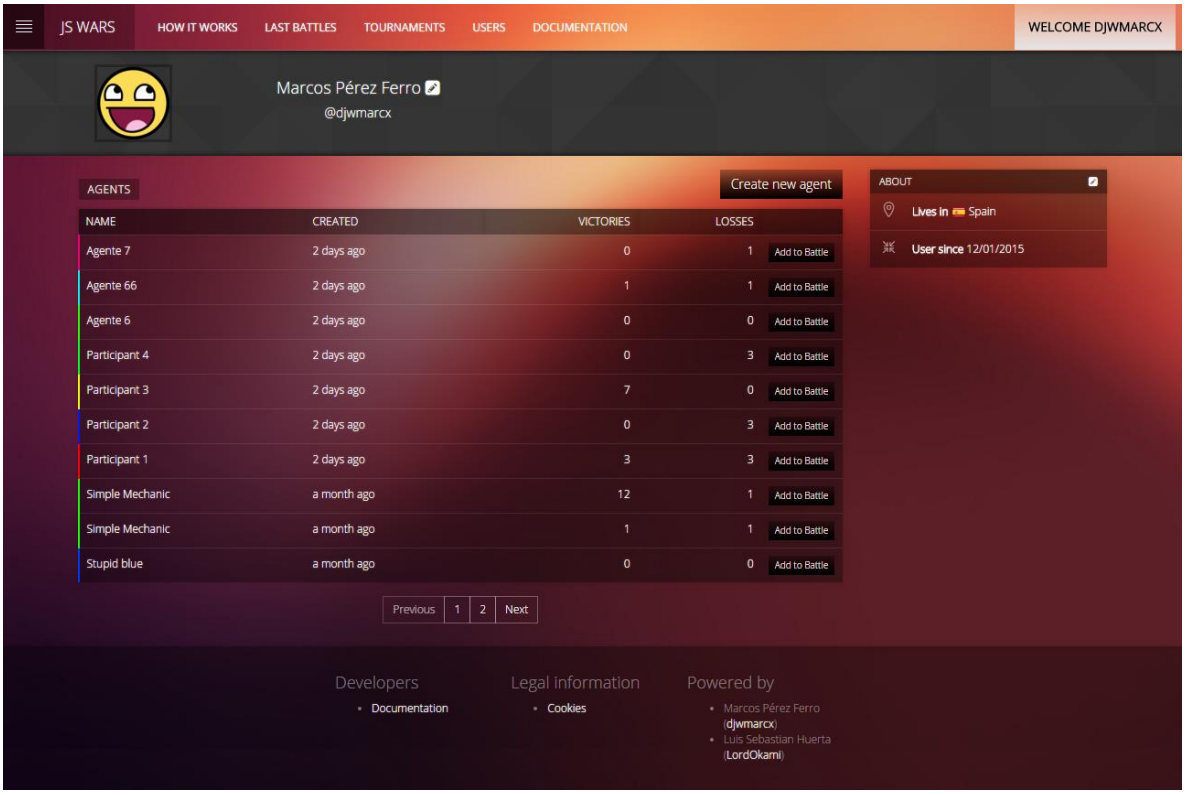


Ilustración 44 *Aspecto de la interfaz 'Detalle de usuario'*

Esta interfaz tiene dos funciones:

- Permite obtener información de cualquier usuario a cualquier usuario (no es necesario tener sesión iniciada)
- Permite gestionar la cuenta propia con sesión iniciada. Es decir, si un usuario entra en el detalle de otro usuario, la verá normalmente, pero si entra en la suyo, podrá crear agentes y editar agentes, además de modificar sus datos de perfil.

5.2.2.9 Crear nuevo agente

La interfaz de crear nuevo agente permite agregar agentes nuevos con todos sus datos.

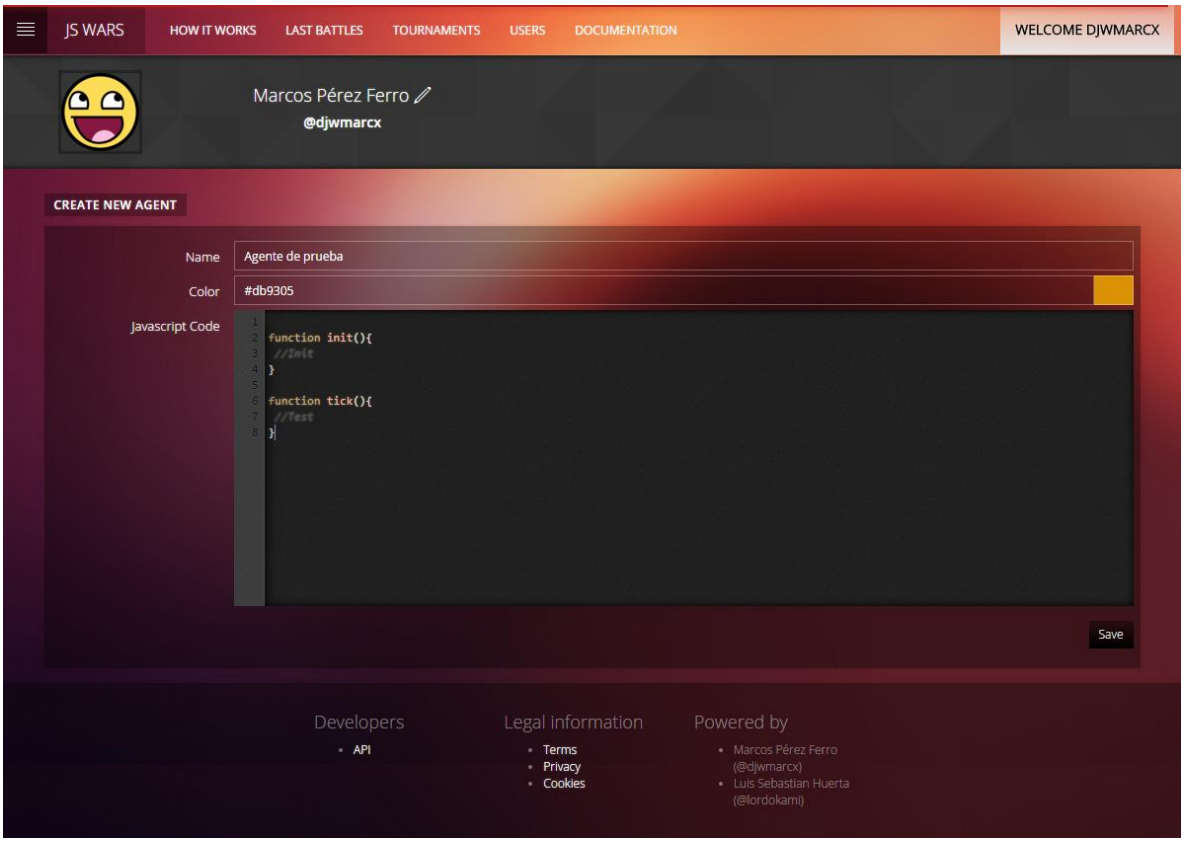


***Ilustración 45** Aspecto de la interfaz 'Crear nuevo agente'*

Los datos que se solicitan para crear un nuevo agente son:

- **Name:** Nombre que tendrá el agente. No se podrá cambiar más adelante.
- **Color:** Color con el que se mostrarán las unidades del agente en las partidas. No se podrá cambiar más adelante.
- **Javascript Code:** Código javascript del agente. Este podrá ser cambiado por el agente tantas veces como se desee.

5.2.2.10 Editar agente

La interfaz editar agente permite a un usuario editar uno de sus agentes ya existentes. Una vez creado un agente, la única información de este que se puede cambiar es el código.

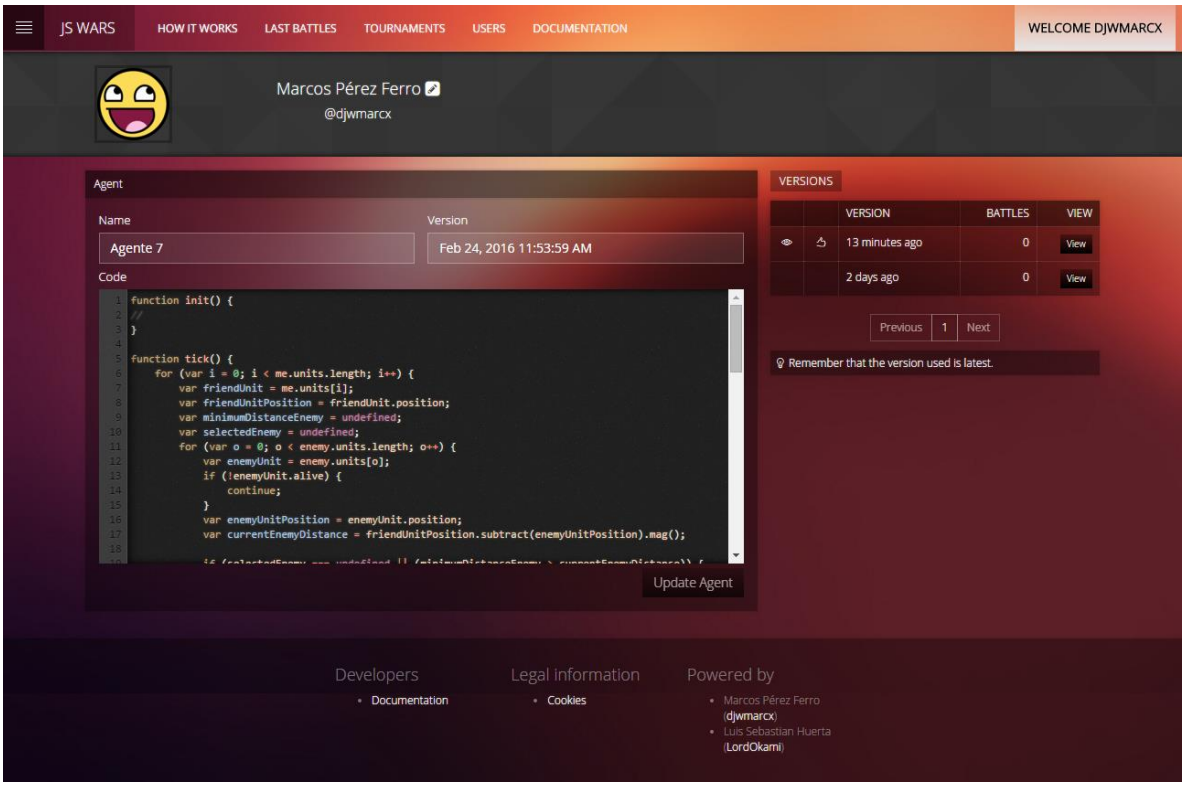


Ilustración 46 *Aspecto de la interfaz 'Editar Agente'*

Para poder explotar la funcionalidad de versionado de código, la pantalla cuenta en la parte derecha con un listado paginado de versiones antiguas que muestra el número de batallas en las que ha participado cada agente y el momento en el que se creó la versión.

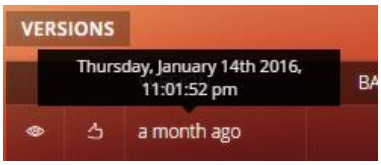


Ilustración 47 *Detalle de hora en el listado de versiones en la interfaz 'Editar agente'*

5.3 CONSIDERACIONES

En este punto revisaremos las soluciones que se han propuesto para ciertos problemas globales del subsistema o específicos con gran peso dentro de este.

5.3.1 Posicionamiento en motores de búsqueda

Cuando la arquitectura web Multiple page application era usada de manera masiva, los motores de búsqueda se moldearon alrededor de esta. La manifestación más evidente de esto es que aún hoy en día, en la era de las Single page application (SPI), a nivel práctico google no indexa nada que no se encuentre disponible en el código del documento principal. Este hecho es una barrera de entrada grande de esta nueva arquitectura web, puesto que ciertos tipos de sitios reciben la mayoría de su tráfico a través de orgánico (Buscadores). Aunque google está trabajando para ejecutar código JavaScript y evitar el esfuerzo de desarrollar herramientas adicionales por parte de los desarrolladores, lo cierto es que pocos datos obtenidos por JavaScript son indexados, entre otros motivos por la complejidad que los desarrollos en JavaScript pueden llegar a alcanzar.

Para solucionar esta problemática con las *Single page application,* han surgido las herramientas de *prerenderizado.* Los sistemas de *prerenderizado* permiten a *google* (Por ahora) obtener una instantánea final de la página incluyendo todo su contenido JavaScript ya procesado. Un entorno con prerenderizado se compone de las siguientes partes:

- **Servidor de recursos estáticos:** En este caso es Apache.
- **Cliente de prerenderizado:** Dentro del servidor de recursos estáticos.
- **Servidor de prerenderizado:** Externo, en este caso *prerender.io*.
- **Buscador web:** Google, Bing, etc…

En el acercamiento clásico de indexación, el motor de búsqueda pide el documento principal de una URL. Como ya hemos comentado anteriormente, las *Single page application* cuentan únicamente con un documento, por lo que:

- Solo se indexa un documento
- El único documento indexado normalmente no tiene contenido.

Para solucionar esto, se han seguido los siguientes pasos (Ejemplificados con AngularJS):

- Activar el formato de URL HTML5: En lugar de cambiar de direcciones con hash (#), se hace con formato de documento nuevo. Cada framework cuenta con su propio método.

  http://domain.com/#/last-battles ⟶ http://domain.com/last-battles

- Agregar *base* y *fragment* al <*head*>: Deberemos añadir los siguientes elementos al documento HTML principal. El primero dice el path sobre el que se debe indexar, y el segundo indica que existe un sistema de prerenderizado disponible.

```
<base href="/">
<meta name="fragment" content="!">
```

En este punto, cuando el buscador acceda al sitio web entenderá, gracias al *fragment* introducido en el documento, que existe un servicio de prerenderizado de HTML y procederá a llamar al documento de la siguiente forma:

http://domain.com/last-battles**?_escaped_fragment_=**

Como se observa, el buscador añade detrás de la URL a indexar un parámetro (Siempre el mismo).

Lo que sucede a partir de aquí es que el *cliente de prerenderizado* diferencia la solicitud de las demás (gracias al parámetro) y en lugar de devolver el documento solicitado en su forma original, le pide al *servidor de prerenderizado* (externo) que se conecte al documento solicitado, procese todo su Javascript como si fuera un explorador y finalmente lo devuelva para que pueda ser enviado al buscador.

En esencia, el *servidor de prerenderizado* actúa como un explorador web para generar todo el documento ejecutando para ello el JavaScript original de la página. El resultado es lo que se envía a google. Este proceso se repite con todas las páginas del mismo enlazadas en los resultados de forma recursiva, por lo que todos los documentos 'virtuales' acaban siendo indexados, habiendo únicamente un fichero *.html en todo el sistema.

5.3.2 Reproducción de partidas y refresco de pantalla

A nivel de sistema, las batallas se ejecutan asumiendo una velocidad de 60 fotogramas por segundo, es decir, 60 ciclos de ejecución de agentes representan 1 segundo de partida. Dentro del elemento <*canvas*> de HTML5, el refresco del fotograma se realiza mediante un evento emitido por el explorador cada vez que el *monitor físico* sobre el que se está mostrando realiza un refresco. La mayoría de monitores generan 60 fotogramas por segundo, con lo que la reproducción de la partida es a la velocidad real. Desgraciadamente, hay ciertos tipos de monitores (como los 3D) que hacen un refresco con más frecuencia, generando más fotogramas por segundo. En estos casos las partidas *no se reproducirían a la velocidad correcta*, si no mucho más rápido (En el caso de los monitores con soporte 3D, exactamente al doble, puesto que generan 120 fotogramas por segundo).

La manera de solucionar esta problemática es no reproducir los *fotogramas* de partida uno tras otro con cada refresco de pantalla. En su lugar, con cada evento de refresco se calcula el tiempo que ha transcurrido desde que se inició la reproducción de la partida para averiguar que *fotograma* le

corresponde al momento actual. De esta forma la reproducción es siempre a la misma velocidad independientemente del refresco de pantalla.

Este cambio además soluciona otro problema común con los gráficos. Trabajar con *<canvas>* requiere uso intensivo de CPU, lo que produce que en ordenadores con una capacidad de proceso limitada los fotogramas tarden más tiempo en dibujarse de lo que realmente hay disponible entre dos fotogramas. Asumiendo que la velocidad real es de 60 fotogramas por segundo:

$$\frac{1\ segundo}{60\ fps} = 0.01667\ s = \ 16.67\ ms$$

Si el tiempo de proceso del fotograma supera los 16.67ms, el siguiente evento de refresco de pantalla no se procesaría y la pantalla quedaría tal cual está hasta el siguiente evento, ralentizando la reproducción de la partida, cosa que como hemos visto, tomando como referencia de fotograma el tiempo de inicio de reproducción, no sucede.

5.3.3 **Recepción de partidas de forma particionada**

Una partida común (1 minuto) tiene alrededor de 3600 fotogramas. Aunque los fotogramas tienen un tamaño relativamente reducido, todos los fotogramas de una partida pueden llegar a pesar más de 6 megas. Por este motivo se ha diseñado un sistema que permite obtener particiones de la partida. Estas particiones están compuestas por defecto por 300 fotogramas que se van solicitando a la vez que la partida va avanzando, antes de que la partición actual termine, asegurando así que la siguiente partición esté disponible cuando haga falta.

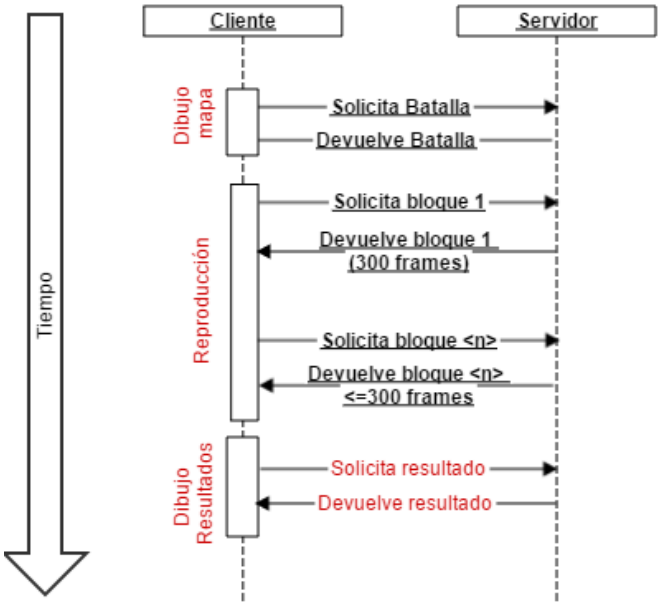


Ilustración 48 *Diagrama de flujo de particionado*

Si todo el conjunto de fotogramas de una partida de 1 minuto pesa **más de 6 megas**, con el particionado cada petición de 300 fotogramas pesa menos de medio megabyte (~ 450kb). Este peso se ve reducido al 10% (**45kb**) gracias a la compresión *GZIP* que se utiliza entre las conexiones del cliente y el servidor, mejorando más aún la experiencia de usuario en dispositivos con conexiones limitadas.

6 SUBSISTEMA SERVIDOR

En este punto vamos a ver todos los detalles de análisis y diseño del subsistema servidor. Este subsistema se ejecuta en el hardware del servidor y contiene toda la lógica de negocio definida a partir de los requisitos y casos de uso. A bajo nivel expone los servicios de la *API Rest* y controla la base de datos, así como el control de la ejecución de las batallas y los torneos.

6.1 EVALUACIÓN DE TECNOLOGÍAS

En este apartado vamos a exponer todas las tecnologías que han sido elegidas para realizar el desarrollo del servidor. En cada una de ellas, se expone brevemente las razones de su elección para este proyecto.

6.1.1 NodeJS

NodeJS es una tecnología de servidor, especialmente diseñada para generar software escalable y mayoritariamente asíncrono. Se apoya en el motor V8 del explorador Google Chrome para ejecutar el código JavaScript, y tiene una gran cantidad de aportes de la comunidad en formato módulos para ampliar su funcionalidad por defecto. Se ha elegido esta tecnología porque en un entorno completamente JavaScript, como es el caso del proyecto actual, la integración con AngularJS y MongoDB es perfecta, y no requiere manejar varios lenguajes.

6.1.2 MongoDB

MongoDB es una base de datos de código abierto NoSQL en la que en lugar de hablar de *tablas* y *tuplas* hablamos de *colecciones* y *documentos*. Las grandes diferencias de MongoDB frente a sistemas clásicos de base de datos es que las consultas se realizan median JSON y que la estructura de los documentos (Lo que serían las tablas en bases de datos clásicas) son muy flexibles, permitiendo por ejemplo definir una columna o atributo del tipo *'Object'* en el que se puede guardar todas la información que se necesite, independientemente de su estructura.

Para este proyecto usaremos MongoDB por el hecho de que tanto las consultas como sus respuestas utilizan JavaScript, lo cual se integra de manera perfecta con NodeJS e incluso con el tipo de respuesta que espera obtener la parte cliente del desarrollo. Su flexibilidad también es muy útil para poder cambiar el comportamiento y objetos presentes en cada fotograma del juego sin alterar la estructura de base de datos, añadiendo mucha flexibilidad a cambios en el formato de partida.

6.1.3 Jshint

JavaScript es un lenguaje muy potente pero con una sintaxis extremadamente relajada. Esto quiere decir, por ejemplo, que se puede escribir código en el que no se inicialicen las variables necesarias (simplemente se usen directamente) y cualquier interprete de JavaScript lo tomará como código perfectamente válido. Desgraciadamente esto que puede parecer cómodo, realmente hace que el código sea más difícil de comprender para el desarrollador e incluso puede desembocar en comportamientos inesperados extremadamente difíciles de localizar.

JSHint es una herramienta de análisis de código desarrollada por la comunidad de JavaScript que busca minimizar al máximo los problemas conocidos de la *relajación* en su sintaxis. Su *modus operandi* consiste en destacar dentro del editor de código esos posibles errores como si fueran errores de sintaxis, evitando así la pérdida de calidad en el código generado.

En este proyecto usaremos JSHint para generar código de calidad y minimizar la posibilidad de errores. Su uso además se verá complementado por el modo estricto de JavaScript (w3schools, s.f.).

6.1.4 Express

Express es un framework para Node.js que tiene como objetivo facilitar la construcción de aplicaciones basadas en *API Rest* (Base de la arquitectura web Single page application (SPI)). El ejemplo más básico de servicio Rest en Express puede no sobrepasar las 10 líneas de código. Como pasa en todos los framework, los proyectos reales requieren mucho más que eso, pero el buen diseño de este se agradece a la hora de usarlo. Una de las cosas más interesantes de Express frente a otros como Restify, es que es sencillo de base y permite ir añadiendo capas de complejidad según va siendo necesario, complicando el sistema solo cuando es estrictamente necesario y sin limitar las capacidades de desarrollo.

En este proyecto se ha elegido Express para evitar tener que dedicar tiempo a desarrollar una arquitectura compleja para gestionar la creación de Rest, pudiendo dedicar ese tiempo a la lógica de negocio, que es lo que realmente aporta valor.

6.2 ARQUITECTURA

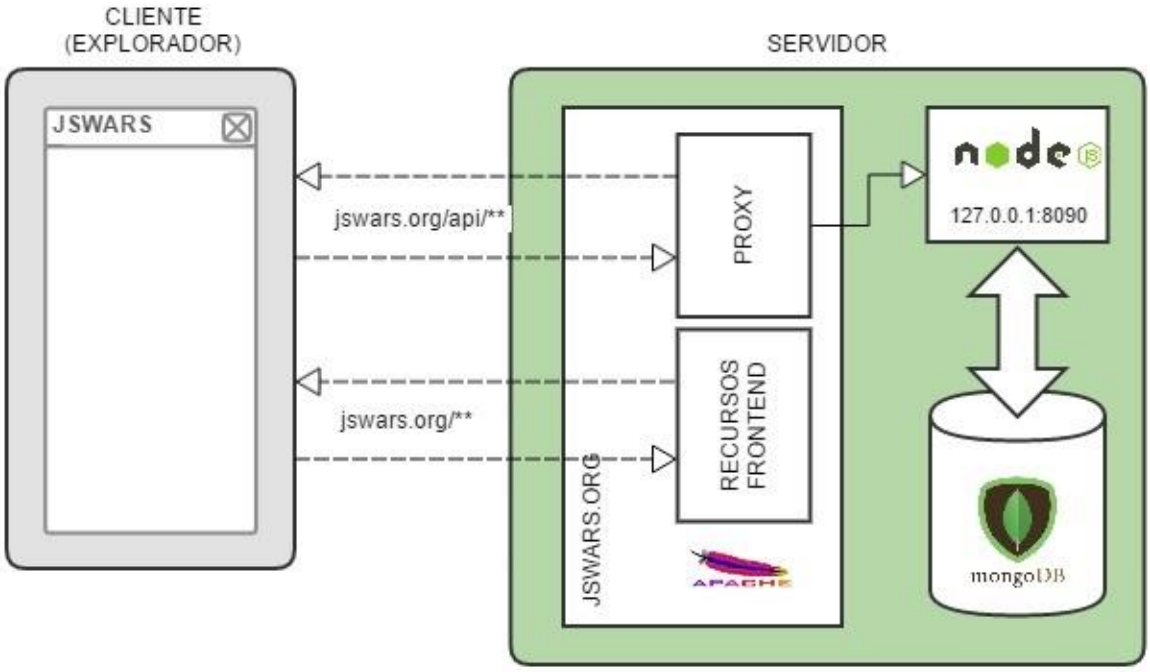


Ilustración 49 *Arquitectura del Servidor*

La arquitectura del servidor posiciona a NodeJS como 'director' de todo lo que sucede en el subsistema. En concreto NodeJS tiene las siguientes responsabilidades:

- Proveer a los clientes del código JavaScript, CSS, etc… del subsistema cliente (Los recursos estáticos)
- Exponer una *API Rest* que permita al subsistema cliente la comunicación con el servidor
- Servir como marco de ejecución del motor del juego y la máquina virtual sobre la que corre.
- Gestionar la *base de datos* (Las consultas y su esquema)

Debido a la arquitectura del conjunto del sistema, el servidor no tiene ninguna responsabilidad en cuanto al aspecto de la interfaz de usuario, limitando sus responsabilidades con respecto al sistema cliente a la *API Rest* y la ejecución de batallas.

La arquitectura se basa en 4 componentes principales que vamos a analizar en los siguientes puntos.

6.2.1 API Rest

En este proyecto, la comunicación entre el cliente y el servidor se produce mediante una *API Rest*. Las *API Rest* funcionan sobre el estándar HTTP y a diferencia de enfoques clásicos, como RPC, la

principal característica es que la definición de los servicios gira entorno a los *recursos* y no a los *métodos*. La semántica de *servicio* se consigue usando combinaciones de métodos HTTP (para identificar el método a llamar) y estados HTTP (Para identificar el resultado de la llamada). Este tipo de implementación aporta una gran semántica y actualmente es un estándar con el que tanto NodeJS como AngularJS son capaces de funcionar sin librerías adicionales. Como detalle, las *API* de este tipo suelen funcionar con JSON, pero también pueden usar XML. (Wikipedia, Representational State Transfer, s.f.)

Operación	*REST*	*RCP*
Obtener Agente	**URL:** /agente/ **Método HTTP:** GET **Estado R.:** 200 (Encontrado)	obtenerAgente()
Crear Agente	**URL:** /agente/ **Método HTTP:** POST **Estado R.:** 201 (Creado)	crearAgente()
Actualizar Agente	**URL:** /agente/ **Método HTTP:** PUT **Estado R.:** 204 (Sin contenido)	actualizarAgente()
Eliminar Agente	**URL:** /agente/ **Método HTTP:** DELETE **Estado R.:** 204 (Sin contenido)	eliminarAgente()

Esencialmente, la definición básica de una operación en RPC es la propia operación en sí (Desconoce sobre que objeto se realizará), mientras que en REST, se crean servicios relacionados con recursos (Lo que en este documento llamamos *Objetos*), y estos servicios cuentan automáticamente con semántica implícita para poder realizar las operaciones básicas sobre el objeto usando como parámetro diferenciador el método HTTP. Estas operaciones suelen ser *leer, modificar, guardar* y *borrar.*

En todos los casos los servicios tienen interacción directa con la base de datos del sistemas, bien para obtener datos, o bien para almacenarlos, salvo *status,* que simplemente devuelve 'OK'.

En la definición actual del proyecto se han diseñado los siguientes servicios con el fin de proveer al sistema cliente de todo lo necesario para poder cumplir los requisitos:

Método HTTP	*URL*	*Objeto*	*Descripción*
GET	/status		Devuelve OK. Permite saber al cliente si el servidor está funcionando correctamente
GET	/session	Usuario	Obtiene el usuario con el que está la sesión iniciada
GET	/login/github	Usuario	Reenvía al usuario al login de GitHub

GET	/login/github/callback	Usuario	Captura al usuario desde el login de GitHub para procesas su nueva sesión
GET	/logout	Usuario	Cierra la sesión actual del usuario
GET	/users/:username	Usuario	Obtiene la información del usuario indicado.
GET	/users	Usuario	Obtiene una lista de usuarios paginada.
PUT	/users/:username	Usuario	Modifica la información del usuario (Solo si el cambio es solicitado por alguien con sesión iniciada en esa cuenta)
GET	/tournaments	Torneo	Obtiene la lista de torneos paginada. (Se puede elegir abiertos o cerrados)
GET	/tournaments/:id	Torneo	Obtiene toda la información del torneo indicado, incluyendo resultados si los hubiese.
POST	/tournaments/:id/join	Torneo	Use al usuario al torneo correspondiente con el agente indicado en el cuerpo de la petición.
GET	/users/:username/agents	Agente	Obtiene una lista paginada de agentes para el usuario especificado
POST	/users/:username/agents/	Agente	Crea el agente definido en el cuerpo de la petición si hay una sesión iniciada con el mismo nombre de usuario. **Se verifica la sintaxis y validez del código.**
GET	/users/:username/agents/:id	Agente	Obtiene el agente con el identificador especificado si hay una sesión iniciada con el mismo nombre de usuario.
PUT	/users/:username/agents/:id	Agente	Modifica el agente con el identificador especificado si hay una sesión iniciada con el mismo nombre de usuario. **Se verifica la sintaxis y validez del código.**
GET	/users/:username/agent/:id/versions	Agente	Obtiene una lista paginada de versiones de agente
GET	/users/:username/agent/:id/ versions/:versionId	Agente	Obtiene la versión concreta de un agente
GET	/battle/queue/:id	Batalla	Obtiene la información de una batalla aún en cola
POST	/battle	Batalla	Obtiene una listado de batallas paginado

GET	/battle	Batalla	Encola una batalla con los agentes definidos en el cuerpo de la petición
GET	/battle/:id	Batalla	Obtiene la información de una batalla con el id solicitado
GET	/battle/:id/chunk/:chunkId	Batalla	Obtiene un fragmento especificado de una batalla
GET	/battle/:id/dump	Batalla	Obtiene un volcado completo de la información básica de la batalla y cada uno de sus fotogramas en formato *JSON.*

Algunos objetos como el *Mapa* no figuran relacionados a servicios puesto que aparecen puesto que no existen operaciones directas contra ellos, si no que se realizar dentro de otros métodos.

6.2.2 Base de datos

Una de las grandes ventajas de MongoDB es que permite guardar *documentos* con atributos sin un formato previamente definido. Es una de las razones por las que se seleccionó MongoDB: De cara a una eventual extensión del motor de juego o la cantidad de mapas soportados, el modelo cambiaría muy poco, puesto que el formato de la información del juego no afecta al diseño del esquema.

El esquema de *base de datos* del subsistema servidor está compuesto por 9 estructuras de datos, relacionadas de la siguiente manera:

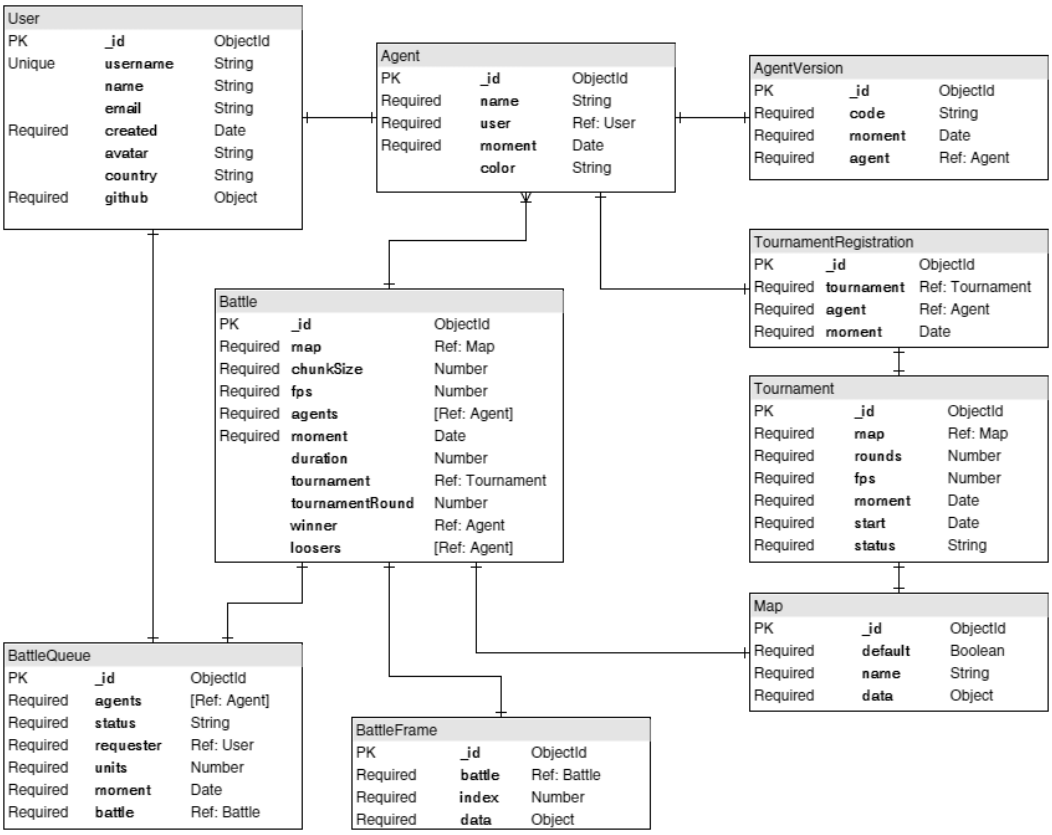


Ilustración 50 *Esquema de base de datos del subsistema servidor*

Las estructuras de datos que podemos encontrar son las siguientes:

- **User:** Contiene la información de los usuarios de la plataforma. La única manera de crear registros nuevos en esta estructura es accediendo con usuarios de GitHub que aún no estén enlazados al sistema. Es la estructura sobre la que se apoyan el resto y cuenta con las siguientes propiedades.
    - *Username*: Nombre usuario en la plataforma.
    - *Name*: Nombre completo del usuario.
    - *Email*: Dirección de correo electrónico.
    - *Created*: Fecha en la que se creó el usuario.

  - *Avatar*: Imagen seleccionada como avatar.
  - *Github*: Contiene toda la información del usuario de Github relacionado con el usuario local.

- **Agent:** Contiene la información de los agentes en la plataforma. Es importante tener en cuenta que esta estructura de datos como tal no contiene nada de código, si no que únicamente sirve como representación de un conjunto de *AgentVersions* que se suponen derivadas unas de otras en cierto orden. Esta estructura aporta al conjunto:
  - *Name*: Nombre del agente.
  - *User*: Referencia al usuario al que pertenece.
  - *Moment*: Día y hora en el que se creó el agente.
  - *Color*: Color con el que se dibujará el agente.

- **AgentVersion:** Representa una versión de código de un agente concreto. Tiene las siguientes propiedades:
  - *Code*: Código de la versión.
  - *Moment*: Día y hora en el que se creó la versión del agente.
  - *Agent*: Referencia al agente al que pertenece.

- **Map:** Contiene la información de un mapa. Tiene las siguientes propiedades:
  - *Default*: Tiene valor *true* si es el mapa por defecto para todo el sistema.
  - *Name*: Nombre del mapa.
  - *Data*: Datos del mapa. Los datos pueden tener cualquier estructura o tamaño.

- **Battle:** Contiene la información de una batalla ejecutada en el sistema. La única manera de crear una batalla en el sistema es mediante *QueueBattle*, que presenta una solicitud de ejecución de una batalla. Las propiedades que contiene son las siguientes:
  - *Map*: Mapa sobre el que se ejecutó.
  - *ChunkSize*: Tamaño del particionado de la batalla. Para más información, revisar Recepción de partidas de forma particionada.
  - *Fps*: Fotogramas por segundo a los que se ejecutó la partida.
  - *Agents*: Agentes que han participado es la ejecución de la partida.
  - *Moment*: Fecha y hora de la ejecución de la batalla.
  - *Duration*: Duración de la batalla en fotogramas.
  - *Tournament*: Campeonato al que pertenece la batalla.
  - *TournamentRound*: Ronda del campeonato al que pertenece la batalla.
  - *Winner*: Referencia al agente ganador de la batalla.
  - *Loosers*: Referencias a los agentes perdedores de las batalla.

- **BattleFrame:** Representa un fotograma de la batalla a la que pertenece. La estructura contiene las siguientes propiedades.
  - *Battle*: Referencia a la batalla a la que pertenece el fotograma.

  - *Index*: Índice del fotograma en el contexto de la batalla a la que pertenece. Este dato es importante para poder hacer una reproducción lineal.
  - *Data*: Contiene la información del estado del juego en el fotograma.

- **BattleQueue:** Representa una solicitud para ejecutar una batalla lo antes posible. Las solicitudes de batalla registradas en esta estructura son ejecutadas por el hilo dedicado a ello. Sus propiedades son:
  - *Agents*: Referencias a los agentes que se desea que participen en la batalla
  - *Status*: Estado de la solicitud. Puede ser:
    i. **PENDING:** Pendiente de ejecutar.
    ii. **RUNNING:** Ejecutando.
    iii. **ENDED:** Ejecución terminada.
    iv. **ERROR:** Falló la ejecución de la partida.
  - *Requester*: Referencia al usuario que solicitó la batalla.
  - *Units*: Número de unidades que debe tener cada equipo en la batalla.
  - *Moment*: Momento en el que se solicitó la batalla.
  - *Battle*: Referencia a la batalla creada partir de esta solicitud. Esta referencia se completa una vez que se crea la batalla.

- **Tournament:** Almacena la información de un torneo. Contiene las siguientes propiedades.
  - *Map*: Referencia al mapa sobre el que se realizarán la batallas del torneo.
  - *Rounds*: Número de rondas del torneo. También determinará el número máximo de participantes.
  - *Status*: Estado de conjunto del torneo. Puede ser:
    i. **PENDING:** Pendiente de ejecutar.
    ii. **RUNNING:** Ejecutando.
    iii. **ENDED:** Ejecución terminada.
    iv. **ERROR:** Falló la ejecución de la partida.
  - *Fps*: Fotogramas por segundo a los que se ejecutará el torneo.
  - *Moment*: Momento en el que se creó el torneo.
  - *Start*: Fecha y hora en la que se comenzará con la ejecución del torneo. Es opcional, si no se especifica, el torneo comenzará cuando se alcance el máximo de participantes. (Determinado por los rounds)

- **TournamentRegistration:** Representa el registro de un agente en un torneo. La única restricción es que no se pueden registrar dos agentes del mismo usuario al mismo torneo. Tiene las siguientes propiedades:
  - *Tournament*: Referencias al campeonato al que pertenece la inscripción.
  - *Agent*: Agente al que referencia la inscripción.
  - *Moment*: Fecha y hora en la que se realizó la inscripción.

Sobre la *base de datos* existen dos *disparadores (trigger)* que desempeñan un papel crucial en el funcionamiento del sistema. Hablamos de los siguientes:

- **TournamentRegistration**: Esta estructura de datos cuenta con un disparador dentro de NodeJS que se dispara cada vez que un agente es registrado para un torneo y notifica tal hecho al módulo *TournamentRunner* (Responsable de ejecutar torneos) para que lance la ejecución del torneo si es necesario, es decir, si se ha alcanzado el máximo de jugadores.
- **BattleQueue**: Esta estructura de datos cuenta con un disparador dentro de NodeJS que notifica al módulo *QueueRunner* (Responsable de ejecutar batallas sueltas) cada vez que se añade una batalla a la cola de ejecución. El módulo es responsable de ejecutarlas de manera ordenada.

En referencia a los disparadores, es importante tener en cuenta que *MongoDB* no cuenta con este tipo de concepto como tal. En su lugar la implementación de estos se hace por software desde *NodeJS*, siendo esto mejor solución puesto que permite mantener las responsabilidades separadas, y en ningún caso se delega ninguna función lógica a la base de datos.

6.2.3 Cola de ejecución de batallas (QueueRunner)

Cuando el usuario solicita una batalla desde la interfaz *Creación de batalla*, lo que sucede es que esta queda almacenada en una estructura de datos dentro de la base de datos llamada *BattleQueue* (cola de batallas), en lugar de ejecutar de manera directa.

La motivación de este diseño es permitir un mejor manejo de las ejecuciones de partida en tiempo real. Actualmente hay poca carga de ejecución de batallas y la CPU disponible es capaz de ejecutar varias de ellas a la vez, pero estos recursos de CPU no pueden ser repartidos de manera ilimitada, por lo que es necesario mantener una cola de ejecuciones pendientes que permita retrasar la ejecución de algunas de las batallas si fuera necesario. Para ello la cola de batallas almacena la información básica de la batalla, como pueden ser los agentes o la cantidad de unidades que lucharán y un campo ‘status’ (Estado) que permite saber en qué estado se encuentra la solicitud:

i. **PENDING:** Pendiente de ejecutar.
ii. **RUNNING:** Ejecutando.
iii. **ENDED:** Ejecución terminada.
iv. **ERROR:** Falló la ejecución de la partida.

Los problemas que soluciona la siguiente implementación con respecto a la ejecución directa son:

- Permite gestionar de manera eficiente gran cantidad de solicitudes de batalla al mismo tiempo sin perder ninguna de ellas.

- Permite que el cliente recupere el estado de una solicitud después de un tiempo (si recarga la página por ejemplo), sin perder por ello visibilidad del estado de la ejecución de la batalla.
- Evita la pérdida de solicitudes de batalla en caso de caída del servidor o fallo del software.

Dado que JavaScript es monohilo, se ha creado un hilo de NodeJS distinto del hilo principal (El que gestiona los servicios de la API Rest y la BBDD) para evitar que la ejecución de las batallas afecte a a los tiempos de respuesta de los servicios, y por lo tanto a la experiencia de usuario.

6.2.4 **Cola de ejecución de torneos (TournamentRunner)**

Cuando se alcanza el número máximo de participantes en un torneo, este módulo recibe una notificación desde un disparador de base de datos y lanza la ejecución del torneo. A diferencia del *QueueRunner*, este módulo no cuenta con ninguna estructura de datos adicional en la base de datos, puesto que ya existe una referencia permanente a los torneos y los agentes registrados en las estructuras de datos *Tournament* y *TournamentRegistration.*

Cuando se recibe y verifica correctamente la solicitud de ejecución de torneo, se lanzan paralelamente todas las batallas de una misma ronda y se espera a que finalicen todas. Una vez que la ronda ha terminado, se recopilan los agentes ganadores y se ejecuta la ronda siguiente teniendo en cuenta solo a los ganadores de la anterior. Este proceso es recursivo y se repetirá tantas veces como rondas tenga el torneo.

Una vez que se alcance la primera ronda, el *ganador del torneo será el que gane la única batalla de la ronda 1.*

Para llevar el control del estado de las batallas y el torneo, se va actualizando el campo *status* de las estructuras de datos *Tournament* y *Battle.* Toda esta información es visible desde la interfaz.

6.3 CONSIDERACIONES

En este punto revisaremos las soluciones que se han propuesto para ciertos problemas globales del subsistema o específicos con gran peso dentro de este.

6.3.1 **Gestión de eventos en NodeJS**

NodeJS es un lenguaje completamente asíncrono en el que todas las operaciones que requieran acceso a cualquier tipo de hardware responden fuera del flujo normal del programa, dado que estas son ejecutadas por defecto en un hilo distinto de forma transparente para el programador (Hay que recordar que NodeJS tiene por definición un único hilo, si no cualquier acceso al hardware bloquearía todo el proceso). Cuando el hilo finaliza la tarea que se le encomendó, este contesta

ejecutando una función que se ha definido previamente para manejar el evento de respuesta de esa petición. A la función que se ejecuta después de finalizar una petición asíncrona se le llama comúnmente *callback.*

Para comprender el funcionamiento de este concepto vamos a ver un ejemplo breve de código.

```
var foundBattle;

Agent.findById("566d62de738d6fd4178b3145")
        .exec(function (battle) {
            foundBattle = battle;
            console.log("Primer log:" + foundBattle)
        });

console.log("Segundo log:" + foundBattle);
```

La salida de este programa sería*:*

```
Segundo log: undefined

Primer log: {“id”:” 566d62de738d6fd4178b3145”,”agents”: {…}, …}
```

Como se puede ver en el anterior ejemplo, la *función* dentro del método *exec* se ejecuta en último lugar (Al revés del orden aparente en el código). Esto es porque es una función *callback* que se ejecuta cuando se recibe la respuesta asíncrona de base de datos, como se ve en la ilustración:

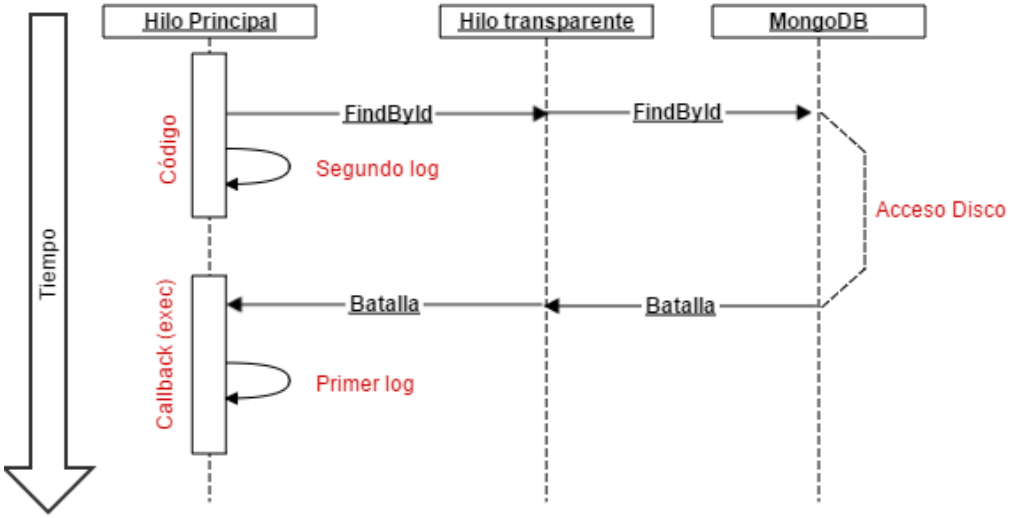


Ilustración 51 *Ejemplo de flujo de eventos y callbacks*

Ahora que hemos visto el ejemplo básico, vamos a ver un ejemplo más complejo. Imaginemos que una petición a un servicio REST en el servidor necesita hacer dos consultas a la base de datos (Sin

relación), y una vez realizadas, dar una respuesta. Para tener todos los datos en el momento de contestar la llamada, el código debería organizarse de la siguiente forma:

```
var getEjemplo = function (request, response) {

    var respuestaCompleta = {
        modeloUno: undefined,
        modeloDos: undefined
    };

    ModeloUno.findById("566d62de738d6fd4178b3145")
            .exec(function (modeloUno) {
                //Obtenemos ModeloUno
                respuestaCompleta.modeloUno = modeloUno;
                ModeloDos.findById("566d62de738d6fd4178b3146")
                        .exec(function (modeloDos) {
                            //Obtenemos ModeloDos
                            respuestaCompleta.modeloDos = modeloDos;
                            //Respondemos a la petición
                            response.json(respuestaCompleta).end();

                        });
            });
};

//Asociamos el método a una petición GET a la ruta
server.get('https://servidor/api/ejemplo/', getEjemplo);
```

Esta implementación funciona correctamente, pero tiene dos problemas grandes:

- **Genera código poco escalable:** Con dos niveles de cascada es posible entender el código, pero con 5 niveles, cualquier editor generará saltos de línea suaves y será casi imposible entender el código. Además la mitad inferior del código estará llena solamente de llaves y paréntesis
- **La ejecución de las peticiones no se hace en paralelo:** Para que se realice la petición n, la petición $n-1$ tiene que haber finalizado, con la consiguiente pérdida de rendimiento (Recordemos que las peticiones no son dependientes, podrían ejecutarse en paralelo)

El flujo exacto de ejecución de este código seria el siguiente:

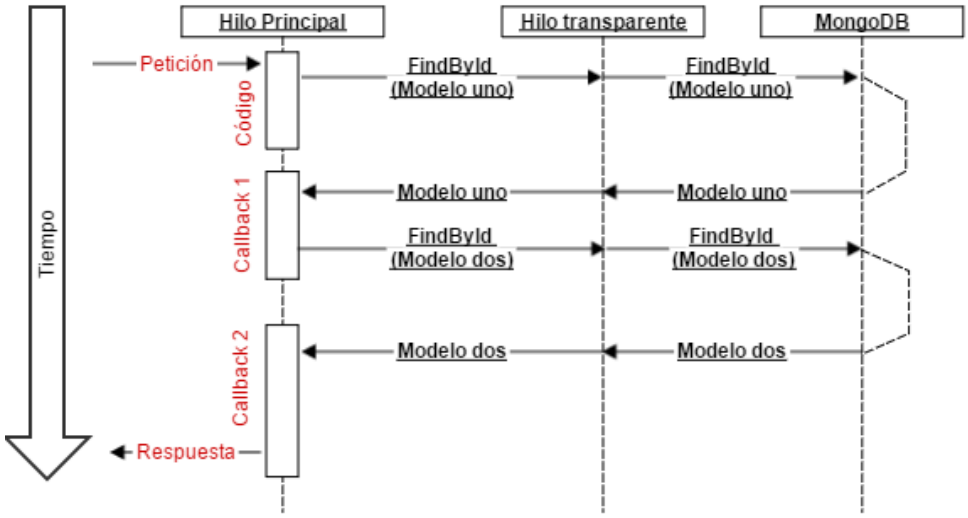


Ilustración 52 *Ejemplo de flujo de multiples callbacks sin promesas*

Para solucionar este tipo de problemática, en JavaScript se hace uso del concepto *Promesa.* Las promesas son esencialmente objetos que son retornados por toda función que ejecuta de forma asíncrona (En las funciones nativas es así, y en las implementadas por el programador debe serlo también, por buenas prácticas). Este objeto permite definir 2 funciones *callback* que serán llamadas cuando se produzcan cambios en el estado de la ejecución asíncrona. Los *callback* son los siguientes:

- ***Resolved* callback:** Es llamado cuando el proceso asíncrono termina de manera satisfactoria. Como primer y único parámetro se recibe el resultado de la ejecución. Por ejemplo, un modelo encontrado en la base de datos.
- ***Rejected* callback:** Es llamado cuando el proceso asíncrono no termina de manera satisfactoria por cualquier motivo. Por ejemplo, no se ha podido conectar con la base de datos

El ejemplo anterior implementado con promesas sería de la siguiente manera:

```
//Importamos nuestra librería de gestión de promesas
var Promises = ("q");

var getEjemplo = function (request, response) {

    var promesas = [];
    var respuestaCompleta = {
        modeloUno: undefined,
        modeloDos: undefined
```

```
    };

    //Solicitamos el modelo uno. 'exec' devuelve una promesa
    promesas.push(ModeloUno.findById("566d62de738d6fd4178b3145").exec());

    //Solicitamos el modelo dos. 'exec' devuelve una promesa
    promesas.push(ModeloDos.findById("566d62de738d6fd4178b3146").exec());

    //Método si todas las promesas son resueltas
    function okHandler(arrayRespuestas) {
        //Respondemos a la petición
        respuestaCompleta.modeloUno = arrayRespuestas[0];
        respuestaCompleta.modeloDos = arrayRespuestas[1];
        response.json(respuestaCompleta).end();
    }

    //Método si alguna promesa es rechazada
    function nookHandler() {
        response.status(500).end();
    }

    //Definimos que pasa cuando se finalizan todas las promesas
    Promises.all(promesas)
            .then(okHandler, nookHandler);
};

//Asociamos el método a una petición GET a la ruta
server.get('https://servidor/api/ejemplo/', getEjemplo);
```

Como se puede observar, se crea un *array* en el que se van añadiendo las promesas (En este caso generadas por las llamadas a la base de datos) y después se registran funciones para manejar los dos casos posibles: *Todas las promesas han sido resueltas (Resolved) y alguna o todas han fallado (Rejected).* El flujo de llamadas queda de la siguiente forma:

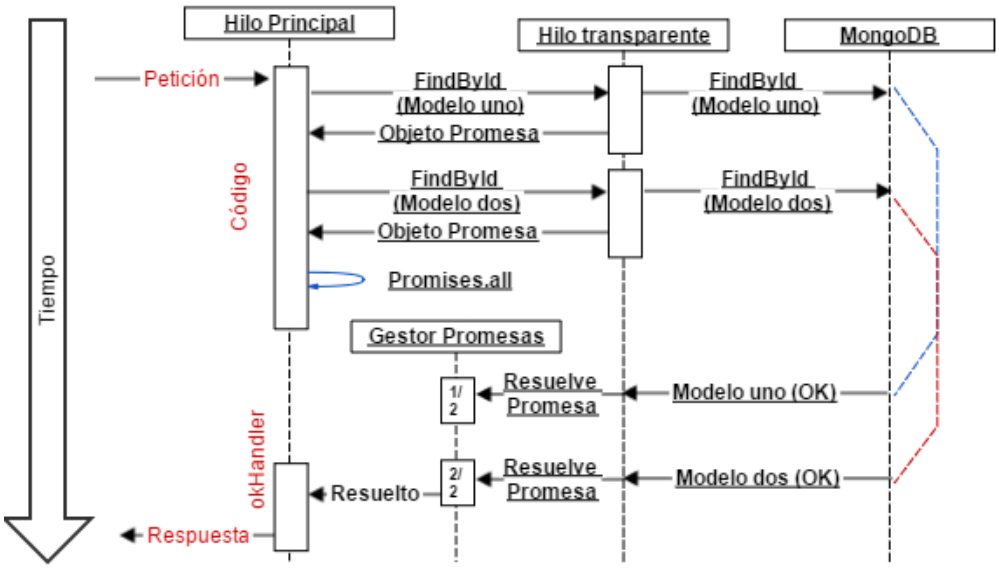


Ilustración 53 *Ejemplo de flujo de multiples callbacks con promesas*

La gran ventaja de este tipo de control de flujo es que se puede llevar un control exhaustivo de múltiples operaciones asíncronas sin perder la capacidad de paralelización de todos ellos.

Las *Promesas* son la base sobre la que se apoya toda la gestión de operaciones asíncronas dentro de este proyecto, puesto que es la única manera eficaz de realizar un proyecto de grandes dimensiones sin sacrificar el rendimiento ni su mantenibilidad.

6.3.2 Almacenamiento en disco

Uno de los retos más importantes de hardware de este proyecto es el control del espacio en disco. Los requisitos del sistema exigen que las partidas puedan ser reproducidas después de haber sido ejecutadas. La ejecución de los agentes inteligentes puede no tener resultados idénticos en ejecuciones bajo las mismas condiciones, por este motivo es necesario almacenar todos y cada uno de los fotogramas de cada partida para su posterior reproducción.

Puesto que el hardware sobre el que se va a montar el proyecto no es ilimitado, vamos a realizar un pequeño análisis de como aumenta el consumo de disco duro (Derivado del uso de la base de datos) en función al número de usuarios, batallas, duración máxima de estas y políticas de expiración o borrado de las partidas.

Duración (Min)	Duración (Seg)	Fotogramas
1	60	3600
2	120	7200
3	180	10800
4	240	14400
5	300	18000

Con el desarrollo del sistema completado, se han realizado pruebas con agentes reales hasta obtener 201 batallas. Estos son los resultados:

Battle Fotogramas	
Fotogramas	196882
Tamaño (Bytes)	211663524
Media (Bytes)	1.075

Battle	
Battles	201
Tamaño (Bytes)	35424
Media (Bytes)	176,238806

En estas ejecuciones, el máximo de *fotogramas* aceptables para una partida estaba situado en 1000. Si se hacen los cálculos, se obtiene que el número de fotogramas medio por partida es de:

$$\frac{196882\ frames}{201\ batallas} \sim 979$$

Evidentemente este valor es muy cercano al maximo, por lo que la primera conclusión que obtenemos es que el valor de 1000 fotogramas como máximo *es muy bajo*. Se asume que el valor

ideal de fotogramas máximo por partida es el equivalente a 1 minuto de partida (Aunque realmente dependerá de la eficacia de los agentes):

$$60\ segundos * 60\ frames = 3600\ frames\ por\ minuto$$

En los primeros meses de la plataforma el nivel de los agentes inteligentes será más bajo, por lo que inicialmente vamos a asumir que la media de tiempo consumido con respecto al tope de un minuto es del 70%, lo que daría:

$$3600\ frames/batalla * 0.7\% = 2520\ frames/batalla$$

Por lo tanto, cada batalla consumirá de media 2520 fotogramas. Teniendo en cuenta el tamaño medio de fotograma obtenido:

$$2520\ \frac{frames}{batalla} * 1.075 \frac{b}{frame} = 2709\ \frac{b}{batalla} * 1024\ \frac{1\ kb}{1024\ b} = 2{,}64\ \frac{kb}{batalla}$$

Dado que el coste de la información de la batalla en si es despreciable y suponiendo que tenemos un disco de 500 GB disponible:

$$500\ gb * \frac{1024\ mb}{1\ gb} = 5120000\ mb * \frac{1024\ kb}{1\ mb} = 5242880000\ kb\ disponibles$$

$$5242880000\ kb * \frac{1\ batalla}{2.64\ kb} = \sim 1985939393\ batallas$$

Con 500 GB dispondríamos de espacio para ejecutar casi dos mil millones, lo cual por ahora es más que suficiente para el uso esperado.

7 HERRAMIENTAS DE DESARROLLO

En esta sección vamos analizar el entorno de trabajo usado para el desarrollo del proyecto. Cuando se inicia el desarrollo del proyecto, la elección de las herramientas desarrollo es muy importante porque estas facilitan (mediante herramientas o funcionalidades) el trabajo de los desarrolladores, permitiéndoles dedicar más tiempo a desarrollar su objetivo. En muchas ocasiones, los requisitos del proyecto pueden convertir una herramienta en mejor opción que otra, por eso es importante hacer un análisis de ellas en cada proyecto.

7.1 SISTEMA OPERATIVO (DEBIAN)

El sistema operativo elegido para el desarrollo del proyecto es Linux (Concretamente Debian). Este sistema operativo cuenta con múltiples ventajas frente a Windows. Es un sistema operativo libre, por lo que no lleva ningún coste asociado a su uso y cuenta con una amplísima gama de soluciones software tanto gratuitas como de pago, para desarrollo web, que es el tipo de desarrollo en el que se engloba este proyecto.

Gran parte de la comunidad de desarrolladores web usa Linux como soporte para desarrollar, lo cual es también un punto positivo de cara a cualquier posible problema durante el desarrollo.

7.2 ENTORNO DE DESARROLLO INTEGRADO (IDE): WEBSTORM

Tanto el cliente como el servidor están desarrollados en lenguaje JavaScript. JavaScript es un lenguaje interpretado que no necesita ser compilado para ejecutarse, por lo que no se necesita un conjunto de herramientas de compilación ni empaquetado , como sucede por ejemplo con Java. Esto abre la puerta a poder usar para su desarrollo simples editores de texto gratuitos como Sublime Text, Atom, Gedit o VIM.

Sin embargo, tras valorar todas las opciones, se ha optado por elegir un software con licencia de pago: WebStorm. Este software pertenece a la empresa JetBrains, caracterizada por desarrollar IDEs muy potentes para facilitar el desarrollo sobre multitud de lenguajes.

La decisión de usar este frente a otros gratuitos, se ha tomado debido a que las herramientas que aporta permiten agilizar en gran medida el desarrollo:

- Autocompletado en HTML5, CSS3 y JavaScript.
- Aviso en tiempo de desarrollo de errores en importaciones y sintaxis en NodeJS
- Debug de JavaScript tanto en explorador como en NodeJS.
- Integración con herramientas de gestión de dependencias como Grunt y Bower.
- Integración con AngularJS.
- Integración con GIT

7.3 CONTROL DE VERSIONES: GIT

Actualmente existe multitud de software que permite el control de versiones de códigos fuente. Algunos de los más populares son SVN, HG (Mercurial) o GIT. En este caso, se ha optado por usar este último. GIT fue desarrollado por el creador de Linux, Linus Torvalds, para la gestión del código del sistema operativo Linux y debe a su origen muchas de sus cualidades. Su potencia y versatilidad lo hace perfecto para casi cualquier desarrollo. Los motivos por los que se ha elegido GIT frente a otro software son:

- Gestión distribuida: Cada usuario cuenta con un repositorio local que sincroniza con el servidor remoto cada vez que hace un pull, push o fetch. El repositorio local puede ser sincronizado con múltiples repositorios remotos e incluso con otros repositorios locales.
- Permite el desarrollo no lineal: Pone disposición de los desarrolladores múltiples herramientas que permiten desarrollar en paralelo en distintas ramas, permitiendo posteriormente hacer mezclas de estas de manera automática o manual. Cada rama puede usarse para evolucionar una funcionalidad distinta, sin que una se vea afectada por los trabajos desarrollados en otra.

7.4 EXPLORADORES

Los desarrollos web tienen la ventaja de que actualmente cualquier dispositivo conectado a internet dispone de capacidad para ejecutarlos. Además no requieren instalación, puesto que todo el código se descarga cuando el usuario accede al servidor que lo sirve por primera vez.

En contrapartida, como el código se ejecuta dentro del explorador del cliente, puede tener comportamientos distintos dependiendo del explorador en el que se ejecute. Existe una cantidad demasiado grande de exploradores y versiones de estos como para asegurar que el desarrollo funcione según lo esperado sobre todos, por lo que las pruebas se van a realizar en Chrome (+41) e Internet Explorer (+10).

8 PLANIFICACIÓN

La planificación para el desarrollo se ha realizado en el periodo anterior a comenzar el conjunto de tareas de análisis y diseño.

Las tareas se han separado en los siguientes grupos representativos:

- ***Análisis y diseño*:** Engloba todas aquellas tareas enfocadas a definir el aspecto, funcionalidades y arquitectura finales del conjunto del sistema. Las tareas dentro de este grupo no generan funcionalidad alguna, por lo que hasta que no se alcance el grupo de tareas de *Desarrollo* no se podrá presentar ninguna funcionalidad al cliente.
- **Desarrollo:** Este grupo contiene todas aquellas tareas que generan funcionalidades tangibles (Se pueden ver y probar). Tanto en este proyecto como en la mayoría de los proyectos de desarrollo, este tipo de tareas se llevan más de un 70% del tiempo total de ejecución del desarrollo.
- **Hitos:** Contiene todos los hitos de entrega a cliente del proyecto. Esta planificación solo cuenta con tres hitos de entrega de funcionalidad, quedando de la siguiente forma:
    1. Presentación del aspecto (Creado por el diseñador). Liberación de toda la gestión de usuarios, incluyendo login de GitHub.
    2. Liberación de la gestión de agentes de usuarios. Ya se pueden crear y modificar guardando cada versión (borrar no es una operación válida). Liberación también de la creación y ejecución de batallas en un mapa.
    3. Liberación del sistema de torneos.

En el diagrama de Gantt se ha decidido no aumentar más el detalle de las tareas puesto que la descripción de las entregas evita que sea necesario (Los hitos de entrega llevan prioridades implícitas).

(El Gantt puede encontrarse en Anexo 1: Gantt Planificación)

La única aportación ágil a la gestión del proyecto es el uso de un tablero *Kamban* (Integrado dentro del gestor de tareas YouTrack). No se han realizado *daily meetings* (Solo dos personas del equipo trabajan en paralelo, es poco útil) y aunque cada una de las entregas podría considerarse *una iteración*, sus tareas no nacen de un *backlog* después de terminar la entrega anterior, si no que están predefinidas desde el principio.

9 ASPECTOS ECONÓMICOS

En este punto vamos a analizar los analizar todos los aspectos económicos relacionados con el proyecto. Primero realizaremos un análisis de los costes de desarrollo, tanto en la parte de *personal* como en la parte de *hardware y licencias*, para terminar analizando el coste estimado de mantenimiento de la plataforma una vez en producción.

9.1 COSTE DE DESARROLLO

En este punto vamos a analizar en profundidad todos los costes necesarios para concluir de manera satisfactoria el desarrollo del sistema. Primero analizaremos los costes del personal teniendo en cuenta sus perfiles, y finalizaremos analizando los costes del *software* y del *hardware* requerido para la realización del proyecto

9.1.1 Personal requerido

Para la realización de este proyecto se ha requerido personal de diversos perfiles. Estos perfiles son:

- **Jefe de proyecto:** Cuenta con la responsabilidad de entender los objetivos del proyecto, realizar una planificación y unos requisitos en base a ellos y garantizar la correcta ejecución de todas las tareas y flujos definidos. La persona que desempeña este perfil es importante que cuente con una buena capacidad de síntesis, mucha facilidad de comunicación con otras personas y una buena capacidad de gestión en situaciones de crisis (Para dar solución problemas a surgidos sin previo aviso y que pongan en peligro la consecución del proyecto). Los estudios ideales para este perfil son: Grado en Ingeniería Informático o equivalente. Es recomendable que supere los 5 años de experiencia como jefe de proyecto.
- **Analista programador:** Su principal responsabilidad es la conversión de los objetivos del proyecto a requisitos de software, además de la definición de la arquitectura de software global del sistema en base a su experiencia y conocimientos. En este perfil es importante la experiencia en desarrollo de sistemas similares, así como la capacidad para conseguir visión global del proyecto sin dejar de lado la escalabilidad del sistema en el futuro (Tanto en funciones como en carga de usuarios). Los estudios ideales para este perfil son: Grado en Ingeniería Informática o equivalente. Recomendable al menos 3 años de experiencia.
- **Programador:** Este perfil tiene como objetivo complementar al analista programador en el desarrollo del código del sistema. En todo momento debe seguir la arquitectura del sistema definida por el Analista Programador y debe informar del estado de sus tareas asignadas periódicamente. Los estudios ideales para este perfil son: Técnico superior en Desarrollo de Aplicaciones o equivalente. Recomendable un año de experiencia.
- **Diseñador:** Será el encargado de asegurar que la experiencia de usuario es lo más agradabale posible y facilita el acceso a todas las funcionalidad de la manera más atractiva

y fácil. Además de funciones de diseño, hará también de maquetador junto al Programador. No existen estudios específicos. Recomendable 3 años de experiencia.

De cada uno de los perfiles se requieren una serie de tareas, que vamos a analizar. En el listado de tareas por perfil, añadiremos además una estimación de dedicación en *horas.*

Perfil	*Tarea*	*Horas*
Jefe de proyecto	Definición de objetivos	56
Jefe de proyecto	Acordar implementación proveedor del motor de juego	24
Jefe de proyecto	Compresión de objetivos y generación de requisitos	24
Jefe de proyecto	Revisión de viabilidad de requisitos con el Analista	8
Analista programador	Análisis de requisitos	24
Analista programador	Elección de tecnologías	40
Analista programador	Definición de la arquitectura de cliente	16
Analista programador	Definición de la arquitectura del servidor	32
Analista programador	Programación del cliente	160
Analista programador	Programación del servidor	280
Diseñador	Diseño del cliente	120
Programador	Programación del cliente	160
Programador	Programación del servidor	240
Total Jornadas		**1184**

9.1.2 **Coste del personal**

Para el desarrollo del proyecto se han dispuesto una serie de salarios brutos por hora y perfil. En base a esto, y teniendo en cuenta la definición de tareas y la estimación de dedicación necesaria, vamos a proceder al cálculo de coste del personal para el desarrollo.

$$Salario\ bruto = \frac{€}{Hora} * Hora$$

Perfil	*Coste/Hora*	*Horas*	*Total*
Analista programador	12,50 €	552	6.900,00 €
Diseñador	8,75 €	120	1.050,00 €
Jefe de proyecto	11,25 €	112	1.260,00 €
Programador	6,875 €	400	2.750,00 €
Total			11.960,00 €

Puesto que el proyecto va a ser realizado durante el 2015 dentro de territorio Español, es necesario tener en cuenta el coste tanto de la Seguridad Social como del IRPF. Teniendo en cuenta el

porcentaje de seguridad social atribuible a la empresa en 2015 es del 23.6%, el coste de empresa es:

$$Coste\ Empresa = Salario\ Bruto * 23.6\%$$

Y usando esta fórmula podemos calcular el gasto real que supone para la empresa la contratación de estos recursos:

Perfil	*Bruto Total*	*SS Empresa (23.6%)*	*Coste Empresa*
Analista programador	6.900,00 €	1.628,40 €	8.528,40 €
Diseñador	1.050,00 €	247,80 €	1.297,80 €
Jefe de proyecto	1.260,00 €	297,36 €	1.557,36 €
Programador	2.750,00 €	649,00 €	3.399,00 €
Total			14.782,56 €

9.1.3 **Costes de Hardware y Software**

En este punto vamos a presentar los costes derivados de las herramientas de trabajo. Comenzaremos analizando el coste directo de los equipos con el software necesario, terminando con un pequeño detalle de costes indirectos generados.

En cuanto al hardware, se han elegido y usado los siguientes equipos (que se van moviendo entre los diferentes recursos según las necesidades):

- **Equipo 1:** PackardBell TS-44HR. Consta de una CPU i5-2410M @ 2.3Ghz 3Mb L3 Cache; Un total de 6GB de memoria RAM; 1 x Grafica GeForce GT 540M (1Gb)
- **Equipo 2:** PackardBell TS-44HR. Consta de una CPU i5-2410M @ 2.3Ghz 3Mb L3 Cache; Un total de 6GB de memoria RAM; 1 x Grafica GeForce GT 540M (1Gb)
- **Equipo 2:** HP ProBook 430 G3 Intel Core i5-6200U/4GB/500GB/13.3"

Equipo	*Precio*
Equipo 1	665.50 €
Equipo 2	665.50 €
Equipo 3	749.00 €
Total	**2080 €**

Una vez definidos los equipos sobre los que se trabajará, es necesario analizar el coste del software necesario para cada uno de ellos. El coste del software en ambos varía debido al software preinstalado con el que cuentan los equipos, e incluso a la configuración elegida de cada uno.

Equipo	*Software*	*Cantidad*	*Precio*	*Total*
Equipo 2	Debian 7	1	0,00 €	0,00 €
Equipo 2	Debian 7	1	0,00 €	0,00 €
Equipo 3	Windows 8.1	1	49.90 €	49.90 €
Equipo 1	WebStorm	(Compartida) 0.5	45,00 €	22,50 €
Equipo 2	WebStorm	(Compartida) 0.5	45,00 €	22,50 €
Equipo 3	Word	1	33,99 €	33,99 €
Total				**128,89 €**

Por tanto, el coste total del *software* y *hardware* asciende a 2208,89 €.

9.1.4 **Costes indirectos**

En todo proyecto de desarrollo, la gran mayoría de los costes suelen venir del personal y de los equipos de trabajo, pero no hay que olvidar que durante el uso de las oficinas, equipos, e incluso en las visitas a proveedores se generan consumos indirectos que hay que tener en cuenta.

En este proyecto, el único gasto indirecto en el que se incurre es el **alquiler de un espacio coworking.** La duración del proyecto está definida y es bastante reducida, por lo que en lugar de buscar y alquilar una oficina se va a optar por un alquiler de un espacio de coworking.

Las ventajas más importantes de este tipo de espacios es que tienen tarifa plana de alquiler en la que *se incluye el consumo eléctrico e internet*. El proyecto se inicia en septiembre y acaba en enero, aunque los días de enero no se alquilará local por ser pocos (Los recursos trabajarán esos días sueltos desde casa). Puesto que el perfil *Diseñador* trabajará en remoto y no acudirá a la oficina en ningún momento, el máximo de personas del equipo simultáneamente en el coworking será de dos.

Teniendo en cuenta que el coste por mes y persona del coworking es de 100 €, los costes indirectos totales ascienden a **800.00 €.** Desplazamientos y dietas corren a cargo de los empleados.

9.1.5 **Resumen de costes**

Tras haber analizado cada uno de los distintos tipos de gastos del proyecto, la tabla sumatoria final y el resultado se puede ver a continuación:

Tipo de coste	*Coste total*
Personal	14.782,56 €
Hardware	2080,00 €
Software	128,89 €
Indirectos	800,00 €
Total	**17.791,45 €**

9.1.6 **Presupuesto**

El coste total generado durante el desarrollo del proyecto es de **17.791,45 €**. Para calcular el presupuesto para el cliente hay que tener en cuenta los *riesgos* y el *margen.* En el presupuesto, estos gravámenes y la seguridad social van incluidos en los precios unitarios, puesto que *el cliente no debe conocerlos*.

- **Riesgos:** Todo proyecto de desarrollo implica ciertos riesgos (bajas del personal, problemas en la toma de requisitos inicial, cambios, gastos inesperados, etc…). Para evitar que la rentabilidad de proyecto se vea afectada por factores externos, se añade un porcentaje sobre los costes. El porcentaje fijado en este caso será del 10%.
- **Margen:** Para obtener beneficios, es necesario añadir al presupuesto un porcentaje en base a los gastos iniciales, asegurando así que reportará beneficios. El porcentaje fijado en este caso también será del 10%.

Teniendo en cuenta esto, podemos calcular fácilmente el presupuesto total que debe figurar en la casilla de totalización del presupuesto:

$$((Coste\ Personal + Otros\ Costes) * (1 + Riesgo + Margen)) * IVA = Presupuesto\ total$$

$$(17.791{,}45\ € * (1 + 0{,}1 + 0{,}1)) * 1.21 = \mathbf{25833{,}19}\ \boldsymbol{€}$$

(El presupuesto puede encontrarse en Anexo 2: Presupuesto desarrollo*)*

La aceptación del presupuesto está supeditada a la aceptación del presupuesto ofrecido por el proveedor del motor de juego, facturado aparte como otro desarrollo.

9.2 COSTE DE PUESTA EN MARCHA Y MANTENIMIENTO

Una vez que el desarrollo se ha realizado, es necesaria su puesta en producción. Este proceso conlleva costes adicionales a los de desarrollo. Esta parte del presupuesto es de aceptación opcional, y podrá ser rechazada (con la consiguiente no realización del servicio) si el cliente desea hacer el mismo la puesta en producción.

9.2.1 Personal requerido

El desarrollo funcionará sobre un servidor conectado a internet, por lo que es necesario configurarlo correctamente para evitar riesgos. Para realizar esta configuración necesitamos el siguiente perfil:

- **Administrador de sistema:** Su responsabilidad es entender el funcionamiento del desarrollo y preparar una arquitectura de sistemas que sea lo suficientemente robusta y segura, haciendo especial hincapié en toda la parte de seguridad (IPTables, Snort, etc…). Los estudios ideales para este perfil son: Grado en Ingeniería Informática. Recomendable un 2 años de experiencia en gestión de políticas de seguridad.

Las tareas a realizar por este perfil son:

Perfil	*Tarea*	*Horas*
Admin. Sistemas	Instalación de la máquina	4
Admin. Sistemas	Instalación del software necesario (NodeJs, MongoDB, apache, etc…)	4
Admin. Sistemas	Instalación y prueba del desarrollo	8
Admin. Sistemas	Configuración de seguridad (IPTables, Snort..)	4
Total Jornadas		20

9.2.2 Coste del personal

El salario acordado con el único perfil necesario para la puesta en producción es:

$$Salario\ bruto = \frac{€}{Jornada} * Jornadas$$

Perfil	*Coste/Hora*	*Horas*	*Total*
Admin. Sistemas	16,25 €	20	325,00 €
Total			325,00 €

Puesto que el proyecto va a ser realizado durante el 2015 dentro de territorio Español, es necesario tener en cuenta el coste tanto de la Seguridad Social como del IRPF. Teniendo en cuenta el porcentaje de seguridad social atribuible a la empresa en 2015 es del 23.6%, el coste de empresa es:

$$Coste\ Empresa = Bruto * 23.6\%$$

Y usando esta fórmula podemos calcular el gasto real que supone para la empresa la contratación de estos recursos:

Perfil	*Bruto Total*	*SS Empresa (23.6%)*	*Coste Empresa*
Admin. Sistemas	325,00 €	76.70 €	401,70 €
Total			**401,70 €**

9.2.3 **Costes de Hardware y Software**

Para desplegar el desarrollo, es necesario contratar un servicio de servidor dedicado. En función a las características del programa se han encontrado las siguientes opciones:

Proveedor	*Nombre*	*Procesador*	*RAM*	*Disco*	*Precio (Mes)*
Kimsufi	KS-4A	Core™ i7-920	16 GB	2 TB	21,99 € + IVA
Kimsufi	KS-5	Xeon 2 x E5504	16 GB	2 TB	24,99 € + IVA
Kimsufi	KS-4B	Core™ i5-3570s	16 GB	1 TB	19,99 € + IVA
SouYouStart	E3-SAT-1	Intel Xeon E3 1225v2	16 GB	2x2 TB (RAID)	30,00 € +IVA

El factor más limitante en este desarrollo en cuanto a hardware es la CPU, puesto que las batallas se ejecutan en tiempo real y requieren CPU para su ejecución. Por este motivo, tenemos que elegir un servidor una CPU potente. Si bien la más potente de todas es la que tiene el modelo **E3-SAT-1** de SoYouStart, este servidor es bastante caro y se desperdicia gran cantidad de memoria y disco duro, por lo que vamos a elegir el siguiente más potente, el **KS-4A** de Kimsufi. El coste final MENSUAL del servidor elegido es por tanto: **26,61 €**

En la configuración del sistema de producción no hay ningún software con licencia de pago, por lo que no existe ningún coste de software.

9.2.4 **Resumen de costes**

Tras haber analizado cada uno de los distintos tipos de gastos del despliegue y puesta en producción del proyecto, la tabla sumatoria final y el resultado son:

Tipo de coste	*Coste total*
Personal	401,70 €
Hardware	26,61 €
Total	428,31 €

9.2.5 **Presupuesto**

En esea proyecto de mantenimiento los costes iniciales suponen **428,31 €**, y un coste mensual posterior de **31,93 €.** Al igual que en el presupuesto de desarrollo, se añade un 10% de margen y un 10% de riesgos, y tanto estos como la seguridad social de los trabajos figura dentro de los precios unitarios.

Teniendo en cuenta esto, podemos calcular fácilmente el presupuesto total que debe figurar en la casilla de totalización del presupuesto:

$$((Coste\ Personal + Otros\ Costes) * (1 + Riesgo + Margen)) *\ IVA = Presupuesto\ total$$

$$\left(428{,}31\ € * (1 + 0{,}1 + 0{,}1)\right) * 1.21 = \mathbf{621{,}90}\ €$$

(El presupuesto puede encontrarse en Anexo 3: Presupuesto puesta en marcha*)*

10 RESULTADOS

Para poder valorar los resultados de la ejecución del proyecto, vamos a usar como medida la experiencia de usuario, en los siguientes niveles:

- Nivel objetivo: Medida de los tiempos de respuesta en las peticiones a servicios y carga de distintas interfaces.
- Nivel subjetivo: Medida de la satisfacción de experiencia del usuario final mediante pruebas con personas ajenas al proyecto.

10.1 PRUEBAS OBJETIVAS

Para medir la calidad objetiva del sistema, vamos a medir los siguientes puntos:

- **Tiempo de carga total de la primera visita (Carga sin caché):** Si es la primera vez que el usuario accede a la aplicación web, el usuario no tiene en la caché del explorador ninguno de los recursos necesarios. Vamos a medir el tiempo de carga total del documento y todos sus recursos en esta primera carga sin caché.
- **Tiempo de carga total de visitas posteriores (Carga con caché):** Si el usuario ya ha entrado al menos una vez a la aplicación web, el usuario dispondrá en su navegador de copias cacheadas de todos los recursos necesarios. Este será el caso típico.
- **Tiempo de respuesta de los servicios:** Esta medida permite valorar por separado la velocidad de respuesta de los servicios web, la cual impacta directamente en el resto de medidas y por tanto en la velocidad percibida por el usuario. Esta medida permitiría detectar en caso de lentitud, el origen de esta.

Para la toma de medidas objetivas, se ha usado el *Inspector web* del navegador C*hrome*. Además las medidas se han tomado en las interfaces más representativas del sistema, para tener una mejor visión de los resultados globales, y los datos expuestos en cada una son medias provenientes de múltiples intentos. Todas las pruebas se han realizado en un ordenador.

Medida	*Listado Batallas*	*Visualización Batalla*	*Listado Usuarios*	*Detalle Usuario*	*Listado Torneos*	*Detalle Torneo*
Carga (Sin caché)	1.13 s	1.96 s	1.40s	1.20 s	1.09 s	1.22 s
Carga (Con caché)	0.752 s	1.30 s	0.828 s	0.816 s	0.780 s	0.830 s
Respuestas de servicios	0.101 s	0.566 s	0.092 s	0.303 s	0.151 s	0. 106 s

** Las medidas pueden diferir entre distintos dispositivos puesto que influyen factores como la versión del navegador, la memoria RAM disponible y la CPU.*

Los tiempos obtenidos en las pruebas resultan altamente satisfactorios, puesto que la percepción del usuario es de carga prácticamente inmediata. Como referencia, la plataforma *CodeFights* comentada en el punto Estado de la cuestión, tiene un tiempo medio de carga con caché de **5.75 ssegundos**.

En el caso de visualización en un dispositivo móvil, sí que se percibe algo más el proceso de carga, pero el resultado sigue siendo bastante bueno.

10.2 PRUEBAS SUBJETIVAS

En este punto vamos intentar medir la satisfacción de los usuarios con las interacciones diseñadas e implementadas.

Para valorar la satisfacción de usuario, se seleccionaron 3 personas ajenas al proyecto, se les proveyó de un agente ya programado (para que no fuera necesario aprender a programarlo) y se les pidió que participaran en 2 batallas con el código de agente provisto. Los usuarios pueden ser encontrados en la plataforma online bajo los siguientes nombres de usuario:

- **Ladamadelascebollas**
- **i1756551**
- **kolmax89**

Todos los usuarios completaron la tarea con éxito, lo cual es muy positivo, pero sí que hubo dos conclusiones comunes a todas las pruebas:

- Hubo cierta tendencia a la *deambulación* al crear el agente. Todos los usuarios esperaban un botón después del editor de código que les permitirá luchar directamente, el cual no existe. Actualmente solo figura un botón que permite actualizar el código del agente. Sería necesario buscar una solución que permitiera añadir directamente el agente a la batalla.

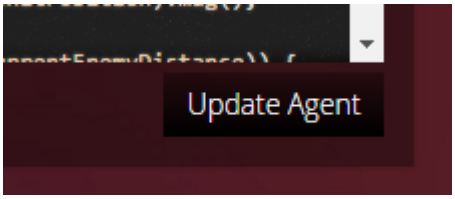


Ilustración 54 *Detalle botón para actualizar agente*

- En ocasiones, los usuarios tenian dificultades para descubrir como volver a su página de usuario. Actualmente, una vez se ha iniciado la sesión, en la parte superior derecha hay un botón que permite ir a la página de usuario, pero en lugar de eso, las personas de las pruebas iban a la lista de usuarios y *se*

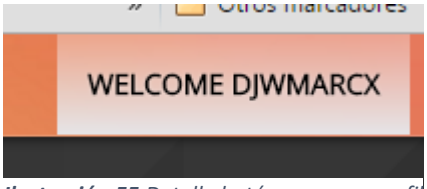


Ilustración 55 *Detalle botón acceso a perfil de usuario*

buscaban en ella. Obviamente, hay que mejorar este caso de uso.

En rasgos generales, podemos decir que el software ha dado buen resultado y cumple con los objetivos para los que se diseñó.

11 LÍNEAS FUTURAS

En este punto vamos a dar un breve repaso de todas las ideas o propuestas que no se han podido implementar en la primera fase del proyecto.

11.1 ESCALABILIDAD

Actualmente la plataforma trabaja sobre 3 únicos hilos de proceso: uno general, otro para la ejecución de batallas solicitadas por los usuarios y uno último para la ejecución de torneos y todas las batallas que estos incluyen.

Cada uno de los hilos secundarios (Batallas y torneos) realiza cada una de las ejecuciones necesarias de manera *secuencial*, por lo que hasta que no se termina la ejecución de la batalla o torneo anterior, no se empieza a ejecutar el siguiente. A nivel de experiencia de usuario, esto no causa problema, puesto que al usuario se le informa en todo momento de que la ejecución solicitada no ha terminado (sigue en curso), pero a nivel de aprovechamiento de CPU si es un problema, puesto que cada hilo ejecuta en un solo *core* de CPU.

La solución esbozada para maximizar el aprovechamiento de CPU se basa en las herramientas para *clustering* que ofrece nativamente *NodeJS*. En esencia, estas herramientas permiten crear 1 hilo maestro y tantos hilos esclavos como se desee (a priori, lo ideal es tantos como *cores* físicos tenga la máquina sobre la que ejecuta). El hilo maestro se ocuparía de *orquestar* el reparto de tareas entre los hilos *esclavos,* mientras que los hilos esclavos se ocupan de ejecutar las tareas y reportar su estado mediante eventos. (NodeJS, s.f.).

11.2 AUMENTAR NÚMERO DE MAPAS DISPONIBLES

Aunque el *modelo de base de datos* está preparado para manejar mapas distintos en partidas distintas, por ahora se ha limitado la funcionalidad y todas las batallas se ejecutan en un mapa por defecto.

Una de las principales problemáticas con esta línea futura es el hecho de que la interfaz cliente no está preparada para permitir buscar un mapa sobre el que se quiere ejecutar una batalla. Para integrar este cambio, la gran mayoría de trabajo vendría por la parte de diseño para la implementación de estas interfaces.

11.3 MÁS DE 2 AGENTES POR PARTIDA

Como se ha comentado en algún punto anterior, el sistema a nivel de *modelo de base de datos* e incluso de *API Rest* está preparado para aceptar 2 o más agentes a la hora de solicitar o ejecutar una batalla, pero hay ciertas partes del sistema que no lo están.

El primer punto a revisar sería el formato de los mapas. Actualmente la posición en la que salen las unidades es fija, por lo que añadir más unidades (Ya fuera del mismo equipo o de uno distinto) supondría un problema. Para manejar esta casuística de la manera correcta, durante el proceso de diseño del mapa habría que añadir previamente los lugares de partida de las unidades, incluso cual es el número máximo de unidades y equipos aceptables.

Además del formato de mapa, habría que realizar importantes cambios en el funcionamiento actual de la interfaz *Configurador de batallas* de la parte inferior de la página. El diseño actualmente solo cumple su función para dos agentes por batalla.

Como se puede ver, no son cambios extremadamente complicados, pero requieren de cierto tiempo de configuración y programación para *una funcionalidad que no entra dentro de los objetivos.*

11.4 RANKINGS

En el acercamiento inicial se planteó hacer un ranking sencillo basado en la relación entre partidas *ganadas* y *perdidas*. Tras realizar un estudio rápido de esta opción se detectó que resultaba muy poco justa en algunas circunstancias. No es lo mismo perder frente al mejor agente de la plataforma, que perder contra el peor, y este tipo de ranking lo contabilizaría de la misma manera.

Una de las opciones de puntuación más conocidas que se ha valorado como candidata para controlar el ranking de los usuarios es *Glicko 2*. Este método tiene la ventaja de que tiene en cuenta el ranking de *ambos participantes* a la hora de recalcular el ranking tras una batalla, suavizando o amplificando la obtención de puntos de cada uno, y volviendo más justo el sistema.

11.5 POTENCIAR FUNCIONALIDADES SOCIALES

Unos de los objetivos subyacentes del proyecto es la fidelización de los jugadores fomentando la competitividad entre ellos. Como en todo aquello que tiene cierto potencial competitivo, siempre hay personas que les gusta seguir los pasos de otras, o incluso buscan la superación retándolas con la intención de ganarles.

Como línea futura se plantea la posibilidad de establecer el concepto de *amistad* entre dos usuarios dentro de la plataforma, permitiendo la interacción entre ellos mediante:

- Mensajes directos.
- Notificaciones de suscripciones a torneos o batallas.
- Notificaciones de subida de nuevo agente.
- Accesos directos.

Adicionalmente, se plantean también las siguientes funciones para todos los usuarios:

- Notificaciones de batalla / torneo perdido.
- Notificaciones de batalla / torneo ganado.

12 CONCLUSIONES

Una vez puesto en producción, el proyecto ha tenido gran aceptación por parte de compañeros y amigos. Los objetivos propuestos se han cumplido al 100%, pero todavía queda un arduo proceso de mejora de la experiencia de usuario y de ampliación de funcionalidades.

La elección de las tecnologías NodeJS, MongoDB y AngularJS ha demostrado ser un gran acierto, permitiendo al proyecto crecer de manera natural sin más complicaciones de las estrictamente necesarias a nivel de arquitectura; todas ellas han demostrado ser tecnologías que definirán los pasos a seguir en el desarrollo web en unos años.

También, gracias a las pruebas de experiencia de usuario realizadas, ha quedado en evidencia la necesidad real de tener muy en cuenta un análisis de UX (Experiencia de usuario) en cualquier desarrollo web exponga alguna interfaz al gran público.

Una vez presentado este trabajo de fin de grado, el código fuente será subido a GitHub en un modelo OpenSource, y espero que podamos continuar con él con colaboración de la comunidad, pues es un proyecto humilde, pero con mucho potencial.

BIBLIOGRAFÍA

Cameron, J., Ham, N., Hamilton, S., & Mistmanov. (s.f.). *AI Challenge*. Obtenido de AI Challenge: http://ants.aichallenge.org/

Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, E., & Pfeiffer, S. (s.f.). *HTML5*. Obtenido de W3C: http://www.w3.org/TR/html5/

NodeJS. (s.f.). *Cluster NodeJS*. Obtenido de NodeJS: https://nodejs.org/api/cluster.html

Sebastián Huerta, L. (2016). *Investigación y desarrollo de una máquina virtual que permita la ejecución de agentes inteligentes en JavaScript.* Madrid.

w3schools. (s.f.). *JavaScript Use Strict*. Obtenido de w3cschools: http://www.w3schools.com/js/js_strict.asp

Wikipedia. (s.f.). *AngularJS*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/AngularJS

Wikipedia. (s.f.). *Hojas de Estilo en Cascada*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada#Niveles_e_historia

Wikipedia. (s.f.). *HTML*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/HTML

Wikipedia. (s.f.). *Inteligencia artificial*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Inteligencia_artificial#Historia

Wikipedia. (s.f.). *Javascript*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/JavaScript

Wikipedia. (s.f.). *JavaScript*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/JavaScript

Wikipedia. (s.f.). *Representational State Transfer*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Representational_State_Transfer

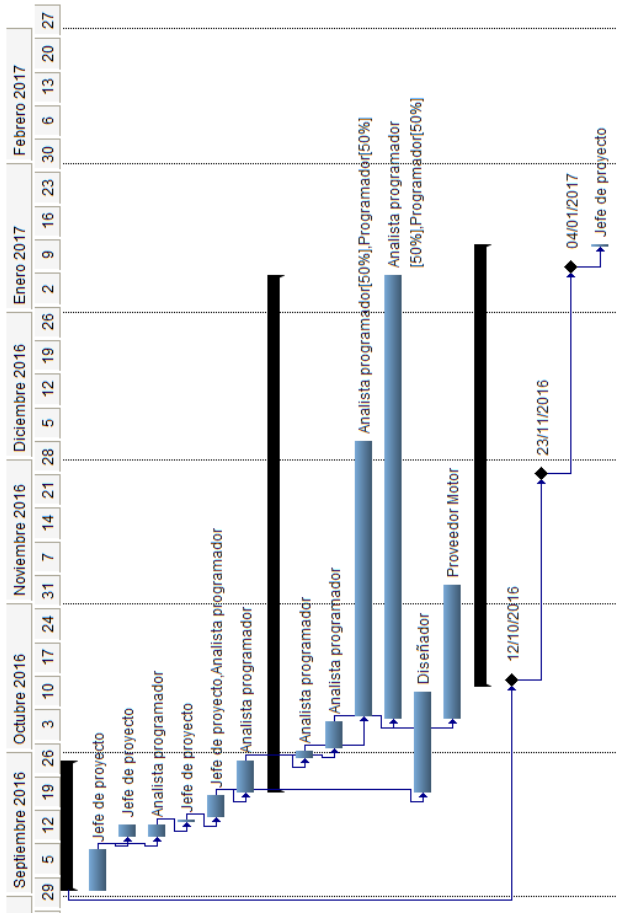
Wikipedia. (s.f.). Single-page application. Obtenido de https://es.wikipedia.org/wiki/Single-page_application

Wikipedia. (s.f.). *Web 1.0*. Obtenido de https://es.wikipedia.org/wiki/Web_1.0

ANEXOS

En las siguientes páginas se pueden encontrar todos los documentos referenciados en el documento principal que no están incluidos en el por su tamaño o forma.

ANEXO 1: GANTT PLANIFICACIÓN



	Nombre	Inicio	Fin
1	**Análisis y diseño**	**01/09/2016**	**27/09/2016**
2	Definición de objetivos	01/09/2016	09/09/2016
3	Acordar implementación proveedor del motor	12/09/2016	14/09/2016
4	Compresión de objetivos y generación de rec	12/09/2016	14/09/2016
5	Revisión de viabilidad de requisitos con el an	15/09/2016	15/09/2016
6	Análisis de requisitos	16/09/2016	20/09/2016
7	Elección de tecnologías	21/09/2016	27/09/2016
8	**Desarrollo**	**21/09/2016**	**04/01/2017**
9	Definición de la arquitectura de cliente	28/09/2016	29/09/2016
10	Definición de la arquitectura del servidor	30/09/2016	05/10/2016
11	Programación del cliente	06/10/2016	01/12/2016
12	Programación del servidor	06/10/2016	04/01/2017
13	Diseño del cliente	21/09/2016	11/10/2016
14	Implementación del motor de juego	06/10/2016	02/11/2016
15	**Hitos**	**12/10/2016**	**10/01/2017**
16	Entrega 1	12/10/2016	12/10/2016
17	Entrega 2	23/11/2016	23/11/2016
18	Entrega 3	04/01/2017	04/01/2017
19	Cierre proyecto	10/01/2017	10/01/2017

ANEXO 2: PRESUPUESTO DESARROLLO

Propuesta comercial formal

Plataforma para gestión de batallas entre agentes inteligentes.

Fecha	Cliente	Proveedor
10 de Diciembre de 2015	Nombre: _____	Marcos Pérez Ferro
	NIF: _____	XXXX
	Dirección: _____	XXXX
	_____	

Formato de pago

Un 30% se pagara al inicio del proyecto, y el 70% restante a su finalización. Supeditado a aceptación del presupuesto externo del motor del juego.

Presupuesto [0001]

Cantidad	Descripción	Precio unitario	Total
552	Hora Analista programador	18.54 €	10.234,08 €
120	Hora Diseñador	12.98 €	1.557,36 €
112	Hora Jefe de proyecto	16.69 €	1.868,83 €
400	Hora Programador	10.20 €	4.078,80 €
1	Gastos de gestión	3.610,67 €	3.610,67 €

Subtotal	21.349,74 €
Impuestos (IVA 21%)	4.483,45 €
Otros conceptos	0 €
Total presupuesto	**25.833,19 €**

Firmado: __*Marcos Pérez Ferro*__, como ___Jefe de Proyecto___.

Este presupuesto tiene una validez de 30 días naturales siempre y cuando esté sellado o firmado por uno de los responsables del proyecto, en caso contrario carecerá de validez alguna.

ANEXO 3: PRESUPUESTO PUESTA EN MARCHA

Propuesta comercial formal

Mantenimiento plataforma para gestión de batallas entre agentes inteligentes.

Fecha	Cliente	Proveedor
10 de Diciembre de 2015	Nombre: _____	Marcos Pérez Ferro
	NIF: _____	XXXX
	Dirección: _____	XXXX
	_____	

Formato de pago

Un 100% al inicio del proyecto e incluye el mantenimiento del servidor el primer mes, a partir de ahí cada mes se pagará solo el coste de *Equipos y Servicios*.

Presupuesto [0002]

Cantidad	Descripción	Precio unitario	Total
20	Jornada Administrador de sistemas	24,10 €	482,04 €
1	Equipos y Servicios	31,93 €	31,93 €

Subtotal	513,97 €
Impuestos (IVA 21%)	107,93 €
Otros conceptos	0 €
Total presupuesto	**621,90 €**

Firmado: __*Marcos Pérez Ferro*__, como ___Jefe de Proyecto___.

Este presupuesto tiene una validez de 30 días naturales siempre y cuando esté sellado o firmado por uno de los responsables del proyecto, en caso contrario carecerá de validez alguna.